Path Tracing Bidirecional em GPU:
Implementação e Análise

Christopher Philippe Diniz Régis

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

João Pessoa, 2019

Christopher Philippe Diniz Régis

Path Tracing Bidirecional em GPU

Monografia apresentada ao curso de Engenharia de Computação do Centro de Informática, da Universidade Federal da Paraíba, como requisito para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Christian Azambuja Pagot
Co-Orientador: Prof. Dr. Andrei Araújo Formiga

Maio de 2019

Catalogação na publicação
Seção de Catalogação e Classificação

R337p Regis, Christopher Philippe Diniz.
Path Tracing Bidirecional em GPU: Implementação e Análise / Christopher Philippe Diniz Regis. - João Pessoa, 2019.
60 f. : il.

Orientação: Christian Azambuja Pagot.
Coorientação: Andrei Araújo Formiga.
Monografia (Graduação) - UFPB/CI.

1. Path Tracing. 2. Path Tracing Bidirecional. 3. GPU. 4. BVH. I. Pagot, Christian Azambuja. II. Formiga, Andrei Araújo. III. Título.

UFPB/CI

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

Trabalho de Conclusão de Curso de Engenharia de Computação intitulado ***Path Tracing Bidirecional em GPU*** de autoria de Christopher Philippe Diniz Régis, aprovada pela banca examinadora constituída pelos seguintes professores:

Prof. Dr. Christian Azambuja Pagot
Universidade Federal da Paraíba (Orientador)

Prof. Dra. Liliane dos Santos Machado
Universidade Federal da Paraíba (Avaliadora)

Prof. Dr. Hugo Leonardo Davi de Souza Cavalcante
Universidade Federal da Paraíba (Avaliador)

João Pessoa, 14 de maio de 2019

Centro de Informática, Universidade Federal da Paraíba
Rua dos Escoteiros, Mangabeira VII, João Pessoa, Paraíba, Brasil CEP: 58058-600
Fone: +55 (83) 3216 7093 / Fax: +55 (83) 3216 7117

"Nem todos os que vagueiam estão perdidos."
(J.R.R. Tolkien)

DEDICATÓRIA

Dedico este trabalho ao meu pai, Gilberto dos Ramos Régis, e a minha mãe, Ana Maria Diniz. Gilberto é um homem sábio e sem egoísmo, que trabalhou desde o dia do meu nascimento para que eu pudesse ter as oportunidades que ele não teve. Ana demonstrou o seu amor por mim todas as vezes em que abriu mão de seus afazeres para me ajudar a alcançar os meus objetivos.

AGRADECIMENTOS

Agradeço ao meu Criador, por ter me dado a oportunidade de viver e por ter dado o seu único filho como sacrifício, para que ao final de toda esta jornada aqui na Terra, eu tenha a oportunidade de passar a eternidade ao seu lado.

Agradeço a minha mãe, Ana Maria Diniz, e ao meu pai, Gilberto dos Ramos Régis, por terem me proporcionado a oportunidade de estudar e por terem me apoiado durante toda a minha vida, em todos os objetivos que eu busquei alcançar.

Agradeço ao meu irmão Anthony Stephen Diniz Régis, por sempre ter estado ao meu lado e me apoiado em todas as situações de alegria e de tristeza, e também por sempre ter estado disponível quando foi preciso.

Agradeço ao meu orientador, Christian Azambuja Pagot, por ter me apresentado o fascinante universo da computação gráfica e por sempre ter me incentivado a evoluir como estudante, como profissional e como pessoa.

Agradeço aos meus irmãos: Lucas Corrêa, Jorgeluis Guerra, Jaelson Carvalho, Gabriel Marques, Elayni Franco, Heronides Laurentino e Lindemberg Falcone, por terem me acompanhado durante parte da minha trajetória na universidade e por muitas vezes terem compartilhado comigo as alegrias e dificuldades de ser um estudante. O tempo ou a distância jamais será capaz de desfazer os laços de amizade que foram construídos.

RESUMO

Hoje em dia cada vez mais pessoas buscam por experiências visuais realistas através de jogos, vídeos ou filmes, onde todos estes produtos consistem basicamente em geração e exibição de imagens. Uma das maneiras de gerar imagens realistas é reproduzir fielmente os efeitos de iluminação que ocorrem no mundo real, como sombras, reflexões, refrações, entre outros. Uma das principais técnicas utilizadas para gerar imagens realistas é o algoritmo de *path tracing*. Porém, a técnica apresenta limitações quando se busca renderizar imagens de cenas onde a maior parte da geometria não é iluminada diretamente pelas fontes de luz. Neste contexto, surge o algoritmo de *path tracing* bidirecional, que funciona como uma extensão do algoritmo original e busca obter melhores resultados. Este trabalho consiste na implementação do algoritmo bidirecional e posterior comparação com outras variações do algoritmo. Os resultados obtidos confirmam que o algoritmo bidirecional é mais eficiente ao renderizar imagens onde a maior parte da geometria não é iluminada diretamente. No entanto, o algoritmo apresenta um maior custo computacional, o que faz com que sua utilização seja inviável em algumas situações.

Palavras-chave: *Path Tracing*, *Path Tracing* Bidirecional, GPU, BVH.

ABSTRACT

Nowadays more and more people are looking for realistic visual experiences through games, videos or movies, where all these products basically consist of generation and display of images. One of the ways to generate realistic images is to faithfully reproduce the lighting effects that occur in the real world, such as shadows, reflections, refractions, among others. One of the main techniques used to generate realistic images is the path tracing algorithm. However, the technique has limitations when rendering images of scenes where most of the geometry is not directly illuminated by light sources. In this context, the bidirectional path tracing algorithm appears, which functions as an extension of the original algorithm and seeks to obtain better results. This work consists of the implementation of the bidirectional algorithm and subsequent comparison with other variations of the algorithm. The results obtained confirm that the bidirectional algorithm is more efficient when rendering images where most of the geometry is not directly illuminated. However, the algorithm presents a higher computational cost, which can make its use unfeasible in some situations.

Keywords: Path Tracing, Bidirectional Path Tracing, GPU, BVH.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

AABB - Axis-Aligned Minimum Bounding Box

BVH - Bounding Volume Hierarchy

CPU - Central Processing Unit

CUDA - Compute Unified Device Architecture

GLSL - OpenGL Shading Language

GPU - Graphics Processing Unit

HLSL - High Level Shading Language

HSV - Hue, Saturation and Value

MBVH - Multi-Branch Bounding Volume Hierarchy

OpenCL - Open Computing Language

OpenGL - Open Graphics Library

RAM - Random Access Memory

SIMD - Single Instruction, Multiple Data

VRAM - Video Random Access Memory

Sumário

1 INTRODUÇÃO

Indústrias como as de *streaming* de vídeos e jogos eletrônicos estão entre as que mais vêm crescendo nos últimos anos (PADRÃO; ALIPRANDI; PORTO, 2018; FLEURY; NAKANO; CORDEIRO, 2014). Os clientes deste tipo de entretenimento buscam cada vez mais por experiências visuais realistas, através de filmes ou jogos cujos efeitos visuais se comprometam a serem os mais parecidos possíveis com o que acontece no mundo real.

Os filmes e jogos eletrônicos são constituídos por várias imagens que são exibidas em uma tela em sequência, cada uma por um curto período de tempo. Por exemplo, os filmes apresentados em uma tela de cinema geralmente são exibidos em uma taxa de 24 imagens por segundo, enquanto os jogos mais atuais exibem cerca de 60 imagens por segundo. Isto faz com que os nossos olhos percebam o que está sendo exibido de forma suave e contínua. Pela forma como todos estes produtos funcionam, pode-se concluir que gerar imagens realísticas implica em gerar filmes e jogos também realísticos.

Uma das maneiras de gerar imagens realísticas é tentar reproduzir fielmente os efeitos de iluminação que ocorrem no mundo real, como os efeitos de sombra, umbra, penumbra, reflexões, refrações, entre outros. Alguns métodos que buscam simular estes efeitos já são utilizados há décadas, como o método da radiosidade (COHEN; GREENBERG, 1985) e o *ray tracing* (WHITTED, 1980). Uma das principais técnicas, que vem sendo estudada e utilizada atualmente para gerar imagens com estes efeitos de iluminação é o algoritmo de *path tracing* (KELLER et al., 2015). A criação deste algoritmo só foi possível devido à formulação da equação de *rendering* realizada por Kajiya (1986).

A equação de *rendering* é uma equação na forma de uma integral e utiliza conceitos de radiometria, que é a ciência que fornece as ferramentas necessárias para compreender e descrever a propagação da luz em um ambiente. O algoritmo de *path tracing* busca resolver computacionalmente a equação através de um método numérico, a fim de determinar quais as cores que os *pixels* de uma imagem devem ter, de modo que os efeitos de iluminação sejam considerados realísticos e fisicamente plausíveis.

O algoritmo de *path tracing* possui limitações ao tentar renderizar imagens de cenas que possuem cáusticas ou onde a maior parte da geometria não é iluminada diretamente pelas fontes de luz (VEACH, 1997; ADAMSEN, 2009). Estas limitações fizeram surgir técnicas de amostragem, que são técnicas que modificam o algoritmo original com o objetivo de obter melhores resultados para as situações onde ele não é eficiente. Neste contexto, as principais técnicas de amostragem são: o método de amostragem direta da luz (SHIRLEY; WANG, 1994) e o *path tracing* bidirecional (LAFORTUNE; WILLEMS, 1993; VEACH, 1997; ADAMSEN, 2009).

O fato dos *pixels* de uma imagem serem independentes entre si, faz com que a renderização de imagens seja um processo naturalmente paralelizável. A busca por melhores desempenhos nestes processos de renderização, fez com que pesquisadores buscassem soluções e implementações em *hardwares* gráficos, devido à quantidade massiva de unidades de processamento que eles possuem (PURCELL et al., 2005; LAINE; KARRAS; AILA, 2013) e também pelo fato dessas tecnologias avançarem rapidamente, tornando o *hardware* cada vez mais eficiente (MISIC; DURDEVIC; TOMASEVIC, 2012). APIs como CUDA e OpenCL possibilitam uma maior facilidade para o desenvolvimento de aplicações que utilizam estes *hardwares* gráficos (COOK, 2013; MUNSHI et al., 2011).

1.1 Objetivo Geral

Este trabalho tem como objetivo desenvolver uma aplicação capaz de executar o algoritmo de *path tracing* bidirecional em GPU (*Graphics Processing Unit*), de modo que seja possível analisar a eficiência deste algoritmo quando comparado ao algoritmo original e ao algoritmo que utiliza o método de amostragem direta da luz. A partir destas comparações, espera-se determinar em quais situações o algoritmo bidirecional é mais eficiente e quando é viável utilizá-lo.

1.2 Estrutura da Monografia

Este trabalho está dividido em 5 seções. A seção 2 apresenta a fundamentação teórica necessária para o entendimento e desenvolvimento do trabalho. A seção 3 apresenta o modo como se pretende desenvolver a aplicação, as escolhas algorítmicas realizadas e a metologia utilizada para os testes. A seção 4 apresenta o modo como aplicação foi estruturada e suas etapas de desenvolvimento. A seção 5 apresenta os resultados obtidos e a análise referente ao desempenho dos algoritmos. Por fim, a seção 6 apresenta as conclusões referentes aos resultados obtidos, os problemas encontrados, as possíveis melhorias e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção apresenta o referencial teórico essencial para a compreensão e desenvolvimento deste trabalho. Inicialmente serão apresentados os conceitos de radiometria e funções de espalhamento da luz, ambos necessários para entender a propagação da luz e o modo como ela interage com as superfícies. Em seguida, serão apresentadas a equação de *rendering* e sua solução numérica através do método de integração de Monte Carlo. Por fim, serão apresentados os algoritmos de *path tracing* e *path tracing* bidirecional.

2.1 Radiometria

A radiometria é a ciência que fornece o conjunto de ferramentas matemáticas necessárias para descrever a propagação da luz e as suas propriedades. Esta subseção apresenta cinco grandezas radiométricas que são essenciais para a formulação e compreensão da equação de *rendering*: energia, fluxo radiante, irradiância, intensidade radiante e radiância. Todos os conceitos que serão apresentados aqui utilizam como base as publicações de Dutre, Bekaert e Bala (2006) e Pharr, Jakob e Humphreys (2016).

2.1.1 Energia e Fluxo Radiante

A luz é uma forma de propagação de energia. Ela é composta por partículas (chamadas de fótons) que transportam quantidades específicas de energia. A energia é a grandeza básica inicial da radiometria e sua unidade de medida é Joules (J).

O fluxo radiante é a quantidade total de energia que passa por uma superfície ou região do espaço por unidade de tempo. Sua unidade é medida em Joules por segundo (J/s), também conhecido como Watts (W). O fluxo radiante (ϕ) é escrito como a razão entre uma unidade de energia diferencial dQ e uma unidade tempo diferencial dt:

$$\phi = \frac{dQ}{dt}.$$

2.1.2 Irradiância

A irradiância é a quantidade total de fluxo radiante que incide sobre a área de uma superfície. Sua unidade é medida em Watts por metro quadrado (W/m^2). A irradiância (E) é escrita como à razão entre uma unidade de fluxo radiante diferencial $d\phi$ e uma unidade de área diferencial dA:

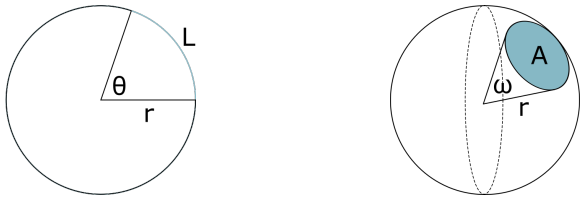
$$E = \frac{d\phi}{dA}.$$

2.1.3 Ângulo Sólido

O ângulo sólido pode ser entendido como uma extensão de um ângulo 2D no plano para um ângulo 3D no espaço. Assim como a medida do ângulo 2D permite determinar o comprimento do arco de uma circunferência, a medida do ângulo sólido permite determinar a área de uma região da esfera (figura 1). A relação entre o ângulo sólido, o raio e a área da esfera é dada por:

$$\omega = \frac{A}{r^2},$$

onde ω representa o ângulo sólido, r é o raio do esfera e A é a área da superfície da esfera que está relacionada ao ângulo sólido ω.



(a) (b)

Figura 1: Exemplos de ângulos. (A) Ângulo 2D, (B) Ângulo sólido.

Para determinar o ângulo sólido de uma superfície arbitrária x em relação a uma área diferencial qualquer x', é preciso projetar a área da superfície x sobre uma esfera ou hemisfério de raio 1, cujo centro é a posição de x' (figura 2). Esta relação é dada por:

$$\omega = \frac{A\,|\cos\theta|}{d^2},$$

onde A é a área da superfície x, θ é o ângulo entre o vetor normal da superfície e o vetor de direção $\overrightarrow{xx'}$. O termo d é a distância entre x e x'.

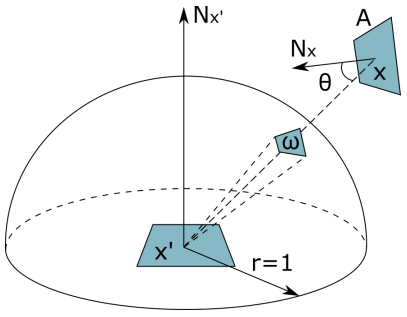


Figura 2: Ângulo sólido de uma superfície arbitrária.

Um ângulo sólido diferencial $d\omega$ pode então ser obtido em função de uma área diferencial dA através da seguinte relação:

$$d\omega = \frac{dA\,|\cos\theta|}{d^2}. \tag{1}$$

2.1.4 Intensidade Radiante

A intensidade radiante é a quantidade de fluxo radiante que atravessa um determinado ângulo sólido. Sua unidade é medida em Watts por esferorradiano (W/sr). A intensidade radiante (I) é escrita como a razão entre uma unidade de fluxo radiante diferencial $d\phi$ e uma unidade de angulo sólido diferencial $d\omega$:

$$I = \frac{d\phi}{d\omega}.$$

2.1.5 Radiância

A radiância é a grandeza que nos informa a quantidade de fluxo radiante que incide ou sai de uma determinada área de superfície, por unidade de ângulo sólido e por unidade de área. Sua unidade é medida em Watts por esferorradiano, por metro quadrado ($W/sr \cdot m^2$). A radiância é escrita como:

$$L(x, \omega) = \frac{d^2\phi}{d\omega \; dA^{\perp}},$$

onde a função $L(x, \omega)$ pode ser entendida como a quantidade de radiância que chega em (ou sai de) uma área diferencial x, pela direção ω. O termo $dA^{\perp}$ é a área diferencial projetada, escrita como:

$$dA^{\perp} = dA \; \cos\theta,$$

onde θ é o ângulo entre o vetor normal da superfície (N_x) e o vetor de direção de incidência ou de saída ω (ver figura 3).

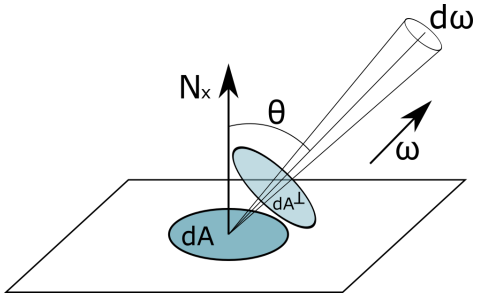


Figura 3: Relação entre a área diferencial projetada $dA^{\perp}$ e o ângulo θ.

A partir da radiância é possível calcular a irradiância total que incide em uma área diferencial x. Para isto, basta somar todos os valores de radiância que chegam por todas as direções ω_i possíveis, pelo hemisfério Ω acima de x (ver figura 4):

$$E(x) = \int_{\Omega} L_i(x', \omega_i) \; |\cos\theta_i| \; d\omega_i,$$

onde $L_i(x', \omega_i)$ é a radiância que sai de alguma área diferencial x' e chega à x pela direção ω_i. O termo θ_i é o ângulo entre o vetor normal da superfície e a direção de incidência ω_i.

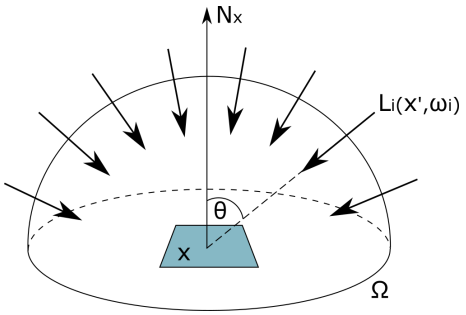


Figura 4: Irradiância que incide sobre uma área diferencial.

2.2 Interação da Luz com Superfícies

A luz interage com as superfícies de acordo com a natureza de seus materiais. Existem materiais que refletem luz, que transmitem luz, que absorvem luz, e na maioria dos casos, materiais que possuem todas estas características juntas. Nesta subseção, serão apresentadas funções que descrevem como a luz se espalha após interagir com as superfícies. Essas funções não só levam em consideração o tipo do material, como também os ângulos de incidência e de saída dos raios de luz que interagem com as superfícies.

2.2.1 Função de Distribuição de Espalhamento Bidirecional

A função de distribuição de espalhamento bidirecional (BSDF) determina qual a relação entre a quantidade de luz que chega em uma superfície x por uma direção ω_i e a luz que sai dela por uma direção ω_o. Em geral, a luz que sai de uma área diferencial x depende da luz que chega a ela por todas as direções ω_i, ou seja, a irradiância total em x. Considerando apenas a contribuição de uma direção particular ω_i, um raio de luz que atinge uma superfície x, contribui com uma irradiância diferencial igual a:

$$dE(x, \omega_i) = L_i(x', \omega_i) \ |\cos \theta_i| \, .$$

Após essa interação, a luz proveniente desta contribuição é espalhada em várias direções (figura 5). Segundo Veach (1997), é possível observar experimentalmente que à medida em que $dE(x, \omega_i)$ aumenta, também há um aumento proporcional nas radiâncias de saída $dL_s(x, \omega_o)$, de modo que:

$$dE(x, \omega_i) \propto dL_s(x, \omega_o).$$

A BSDF é então definida como sendo este termo de proporcionalidade:

$$f_s(x, \omega_o, \omega_i) = \frac{dL_s(x, \omega_o)}{dE(x, \omega_i)} = \frac{dL_s(x, \omega_o)}{L_i(x', \omega_i)\ |\cos\theta_i|}, \tag{2}$$

onde $f_s(x, \omega_o, \omega_i)$ é a função de espalhamento, x é o ponto da superfície com que a luz interagem, ω_i é a direção de incidência e ω_o é a direção de saída.

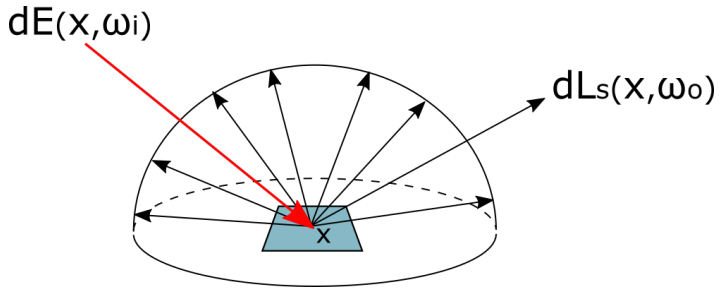


Figura 5: Exemplo de espalhamento da luz após interagir com uma superfície.

A luz proveniente de espalhamento pode ser subdividida em duas componentes: a parte da luz que é refletida e a parte da luz que é transmitida. Estas duas componentes podem ser tratadas separadamente e somadas para compor a BSDF:

$$f_s(x, \omega_o, \omega_i) = f_r(x, \omega_o, \omega_i) + f_t(x, \omega_o, \omega_i).$$

onde f_r é a função de distribuição de refletância bidirecional (BRDF), e f_t é a função de distribuição de transmitância bidirecional (BTDF).

A BRDF e a BTDF são obtidas restringindo o domínio da BSDF. Enquanto a BRDF considera todas as direções de saída somente pelo hemisfério Ω_r, que está no mesmo sentido da normal da superfície, a BTDF considera todas as direções de saída do hemisfério Ω_t, que está no sentido contrário a normal da superfície (figura 6).

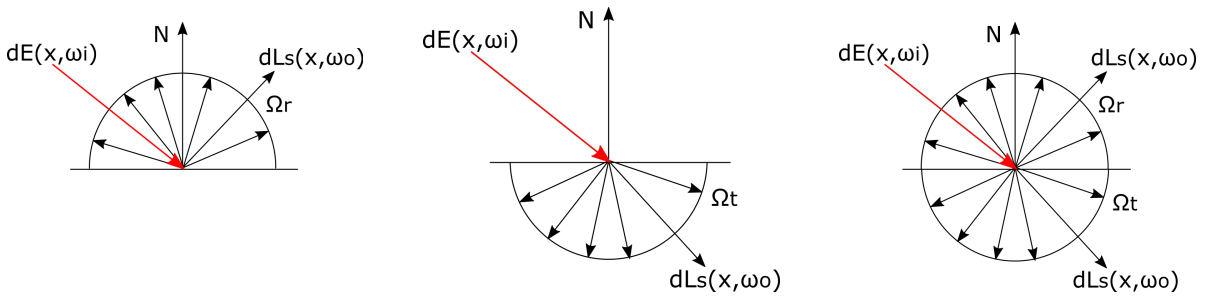


Figura 6: Funções de espalhamento. (A) BRDF, (B) BTDF, (C) BSDF.

2.3 A Equação de Rendering

A equação de *rendering* foi formulada por Kajiya (1986). Ela capaz de determinar a radiância de saída de uma área diferencial x em qualquer direção ω_o. A radiância de saída $L_o(x, \omega_o)$ é definida como a soma entre a radiância emitida $L_e(x, \omega_o)$, caso a superfície seja uma fonte de luz, e a radiância espalhada $L_s(x, \omega_o)$:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + L_s(x, \omega_o), \tag{3}$$

A radiância espalhada por uma superfície x, depende das radiâncias $L_i(x', \omega_i)$ que incidem nela, provenientes de espalhamento de outras superfícies (figura 7).

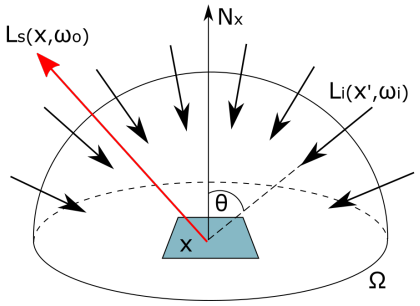


Figura 7: Radiância refletida por uma área diferencial x.

A radiância espalhada é obtida rearranjando a equação 2 e integrando em ambos os lados da equação no intervalo de todas as direções ω_i possíveis:

$$L_s(x, \omega_o) = \int_\Omega L_i(x', \omega_i)\ f_s(x, \omega_o, \omega_i)\ \left|\cos\theta_i\right|\ d\omega_i. \tag{4}$$

Substituindo o termo $L_s(x, \omega_o)$ da equação 3 pela equação 4, obtemos a forma final da equação de *rendering*:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega L_i(x', \omega_i)\ f_s(x, \omega_o, \omega_i)\ \left|\cos\theta_i\right|\ d\omega_i. \tag{5}$$

A formulação da equação 5 é conhecida como formulação em hemisfério ou ângulo sólido, pois o domínio de integração são hemisférios ou ângulos sólidos. Outras duas formulações importantes serão apresentadas nas subseções a seguir: a equação de *rendering* na forma dos três pontos, também conhecida como formulação em área, e a formulação no espaço dos caminhos. A formulação em área considera os domínios de integração como sendo áreas de superfícies, enquanto a formulação no espaço dos caminhos considera os possíveis caminhos (trajetórias) que a luz faz em um ambiente.

2.3.1 A Forma dos Três Pontos

A forma dos três pontos é uma forma que considera áreas diferenciais como sendo pontos e os ângulos sólidos como sendo direções (figura 8). Esta forma foi introduzida inicialmente por Kajiya (1986) e analisada com mais detalhes por Veach (1997). A equação de *rendering* na forma dos três pontos avalia integrais cujos domínios de integração são áreas de superfícies. Esta maneira de representar a equação é utilizada em algoritmos que utilizam amostragem direta da luz, método que será apresentado na subseção 2.5.1. A forma dos três pontos também é utilizada na formulação da equação de *rendering* no espaço dos caminhos, que será apresentada na subseção 2.3.2.

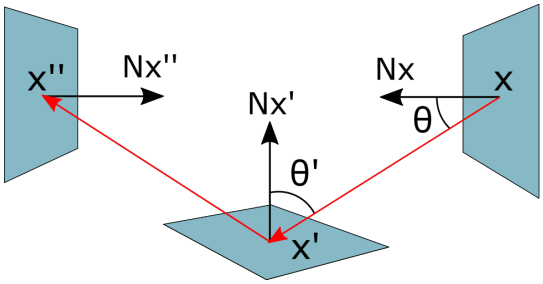


Figura 8: Forma dos três pontos.

Para reescrever a equação 5 como uma integral no domínio da área, é preciso eliminar as variáveis de ângulo sólido ω_o e ω_i. A radiância é então reescrita da forma:

$$L_o(x', \omega_o) = L_o(x', \overrightarrow{x'x''}), \tag{6}$$

onde x' e x'' são áreas diferenciais (ou pontos) e $\omega_o = \overrightarrow{x'x''}$ é um vetor unitário que aponta de x' para x''. Neste caso, x'' é o ponto onde pretende-se avaliar a radiância incidente.

De forma semelhante à radiância, a BSDF é reescrita da forma:

$$f_s(x', \omega_o, \omega_i) = f_s(x', \overrightarrow{x'x''}, \overrightarrow{xx'}). \tag{7}$$

O ângulo sólido diferencial de x em relação a x' pode ser reescrito em função da área diferencial de x (semelhante a equação 1):

$$d\omega = \frac{|\cos\theta|}{\|x - x'\|^2}\, dA. \tag{8}$$

Substituindo as equações 6, 7 e 8 na equação 5, e considerando que a radiância que incide de x' é espalhada na direção de alguma superfície arbitrária x'' (figura 8), a equação de *rendering* na forma dos três pontos fica da forma:

$$L_o(x', x'x'') = L_e(x', x'x'') + \int_A L_i(x, xx')\; f_s(x', \overrightarrow{x'x''}, \overrightarrow{xx'})\; G(x \leftrightarrow x')\; dA, \tag{9}$$

onde $G(x \leftrightarrow x')$ é chamado "termo geométrico", dado por:

$$G(x \leftrightarrow x') = |cos\theta'| \ \frac{|\cos\theta|}{\|x - x'\|^2} \ V(x \leftrightarrow x'), \tag{10}$$

onde $V(x \leftrightarrow x')$ é uma função de visibilidade onde:

$$V(x \leftrightarrow x') = \begin{cases} 1, & \text{se } x \text{ e } x' \text{ são mutuamente visíveis;} \\ 0, & \text{caso contrário.} \end{cases} \tag{11}$$

2.3.2 Espaço dos Caminhos

O espaço dos caminhos Ω representa o conjunto de todos os possíveis caminhos formados pela trajetória da luz, de todos os comprimentos possíveis (VEACH, 1997). Um espaço Ω_k é o espaço que representa todos os caminhos de comprimento k, de modo que:

$$\Omega = \bigcup_{k=1}^{\infty} \Omega_k.$$

Um caminho é representado pelos pontos das superfícies em que a luz interagiu até chegar em um determinado ponto de avaliação. Um caminho $\overline{p_k}$ é escrito da forma:

$$\overline{p_k} = x_k, x_{k-1}, x_{k-2}, ..., x_2, x_1, x_0,$$

onde k é o comprimento do caminho. A figura 9 mostra o exemplo de um caminho com comprimento $k = 3$.

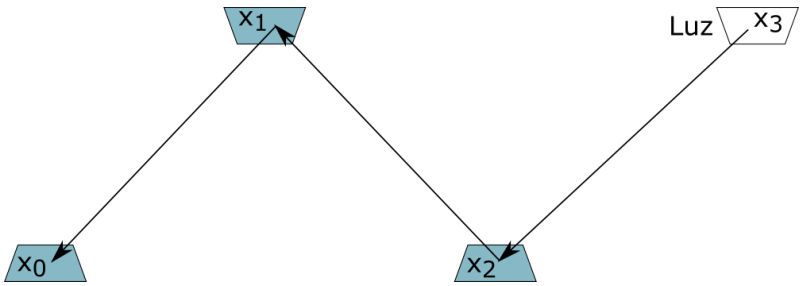


Figura 9: Exemplo de um caminho com comprimento 3.

Para reescrever a equação de *rendering* no espaço dos caminhos, é preciso expandir a equação de *rendering* formulada em área de forma recursiva, utilizando a expansão de integral em série de Newman (ADAMSEN, 2009). A expansão em série de Newman possibilita reescrever uma equação com integral da forma:

$$L(x) = f(x) + \int_A L(x) \ g(x) \ dA,$$

em uma séria na forma:

$$L(x) = \sum_{m=0}^{\infty} \underbrace{\int_A \cdots \int_A}_{m} f(x) \left(\prod_{i=1}^{m} g(x) \right) \underbrace{dA \cdots dA}_{m}.$$

Expandindo a equação 9 pelo método de Newman, temos a equação de *rendering* formulada no espaço dos caminhos em sua forma compacta (ADAMSEN, 2009):

$$L(x_1 \to x_0) = \sum_{i=1}^{\infty} P(\overline{p_i}), \tag{12}$$

onde o termo $L(x_1 \to x_0)$ está de acordo com a figura 9, onde o ponto de avaliação é o ponto x_0. O termo $\overline{p_i}$ representa um caminho de tamanho i e $P(\overline{p_i})$ é uma função que mede a contribuição de cada caminho $\overline{p_i}$, dada por:

$$P(\overline{p_i}) = \underbrace{\int_A \cdots \int_A}_{i-1} L_e(x_i \to x_{i-1}) \left(\prod_{j=1}^{i-1} f_s(x_{j+1} \to x_j \to x_{j-1})\, G(x_{j+1} \leftrightarrow x_j) \right) dA(x_2)...dA(x_i). \tag{13}$$

2.4 Método de Integração de Monte Carlo

Os métodos de Monte Carlo são métodos numéricos que buscam obter resultados de funções baseando-se em amostragens aleatórias. Segundo Veach (1997), o método para integrais possui uma taxa de conversão de aproximadamente $O(N^{\frac{1}{2}})$, onde N é o número de amostras utilizadas. Além de simples, o método permite solucionar integrais nos mais variados domínios de integração.

Considerando uma integral na forma:

$$I = \int_D f(x)\, dx,$$

onde D é o domínio de integração e $f(x)$ é uma função que mede a contribuição de cada amostra $x \in D$. O método integração de Monte Carlo permite aproximar o resultado desta integral, tomando uma média de N amostras aleatórias da seguinte forma:

$$I \approx F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)},$$

onde N é o número de amostras, x_i é o valor da i-ésima amostra e $p(x_i)$ é a função de distribuição de probabilidade (PDF) avaliada em x_i. A medida em que o número de amostras N aumenta, o método se aproxima cada vez mais da solução exata.

2.5 Path Tracing

O algoritmo de *path tracing* busca resolver a equação de *rendering* através do método de integração de Monte Carlo. A equação de *rendering* formulada no hemisfério (equação 5) e estimada com o método de Monte Carlo é dada por:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \frac{1}{N} \sum_{i=1}^{N} \frac{L_i(x', \omega_i)\ f_s(x, \omega_o, \omega_i)\ |\cos\theta_i|}{p(\omega_i)}, \tag{14}$$

O algoritmo de *path tracing* tem como objetivo determinar o valor da radiância que sai de alguma superfície x e vai em direção aos sensores de uma câmera (figura 10). Na grande maioria das vezes, estes sensores representam os *pixels* da imagem que será formada após o término do algoritmo. Os valores de radiância, portanto, determinam as cores dos *pixels* da imagem.

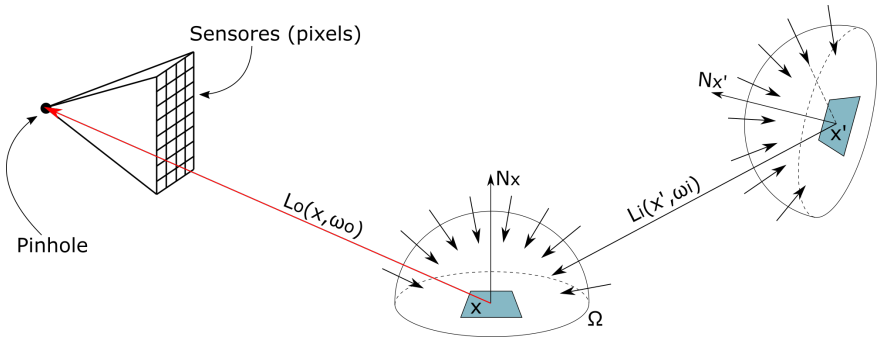


Figura 10: Relação entra a radiância e os *pixels* de uma imagem.

Para obter o valor exato de radiância que cada *pixel* deve receber, é preciso considerar todas as radiâncias incidentes $L_i(x', \omega_i)$ que chegam às superfícies por todas as direções possíveis (figura 10). Deste modo, para resolver a equação de *rendering* através do método de Monte Carlo (equação 14), seria preciso utilizar um número de amostras $N = \infty$. Entretanto, o que se faz na prática é considerar apenas uma amostra de radiância incidente, ou seja, $N = 1$. O que reduz a equação 14 para a forma:

$$L_{o_i}(x, \omega_o) = L_e(x, \omega_o) + \frac{L_i(x', \omega_i)\ f_s(x, \omega_o, \omega_i)\ |\cos\theta_i|}{p(\omega_i)}, \tag{15}$$

com isso, a solução deixa de ser avaliar uma média de N amostras de radiâncias incidentes $L_i(x', \omega_i)$, e passa a ser avaliar uma média de N amostras de radiâncias $L_{o_i}(x, \omega_o)$ por *pixel*, da forma:

$$L_o(x, \omega_o) = \frac{1}{N} \sum_{i=1}^{N} L_{o_i}(x, \omega_o).$$

Pela equação 15, o problema de determinar o valor de uma amostra de radiância que chega em um *pixel*, consiste apenas em simular a trajetória de um único raio de luz. No mundo real, a luz sai da fonte de luz, interage com as superfícies do ambiente e por fim chega até a câmera. O método que simula este caminho é chamado *forward tracing*.

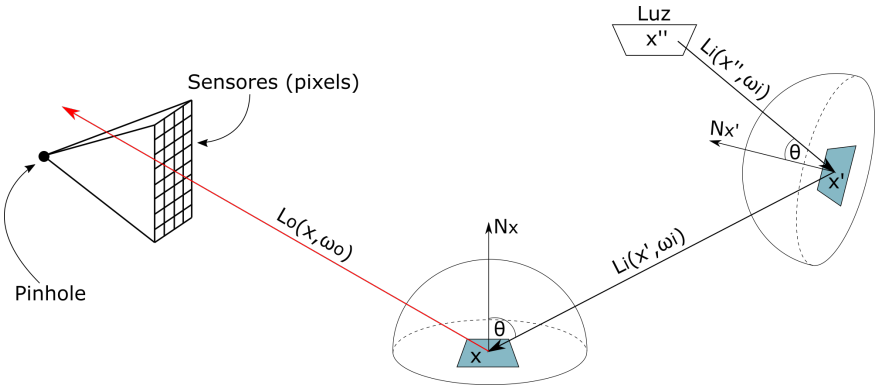


Figura 11: Método *forward tracing*.

O problema do método *forward tracing* é que nem todo raio de luz chega até câmera, principalmente se for uma câmera *pinhole* (figura 11). A melhor maneira de simular a trajetória da luz é fazer o raio partir da câmera, interagir com o ambiente, e por fim, chegar até a fonte de luz (figura 12). Este método é chamado de *backward tracing*. Com ele, garante-se que todo raio que cruza a luz, também cruza a câmera.

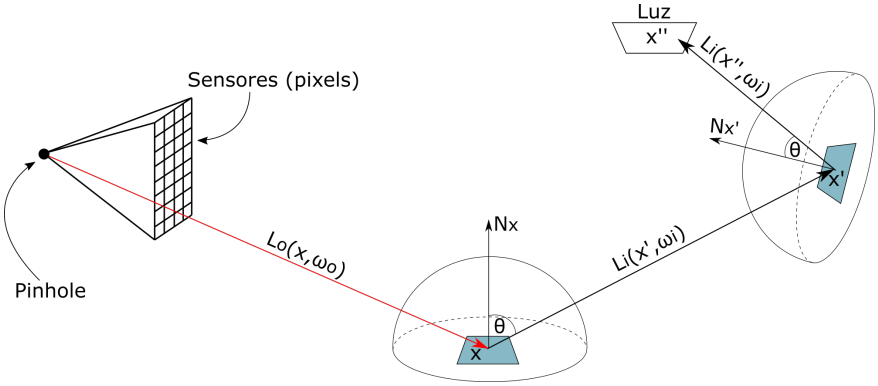


Figura 12: Método *backward tracing*.

2.5.1 Amostragem Direta da Luz

O problema do método *backward tracing* é que apesar de garantir que todo raio que cruza a luz também cruza a câmera, a probabilidade de se atingir um fonte de luz cuja área é muito pequena, também é muito pequena. Deste modo, qualquer processamento gasto para avaliar uma trajetória que não cruza a luz é desperdiçado. Neste contexto, Shirley e Wang (1994) sugerem melhorar o algoritmo de *path tracing* utilizando um método com amostragem direta da luz.

A ideia da amostragem direta da luz é verificar se a luz é visível a partir de cada ponto de interação que a luz realizar com as superfícies (figura 13). Se a luz é visível a partir de um determinado ponto de interação, dizemos que este ponto é iluminado diretamente. Deste modo, o algoritmo é eficiente para ambientes em que a maior parte das superfícies são iluminadas diretamente.

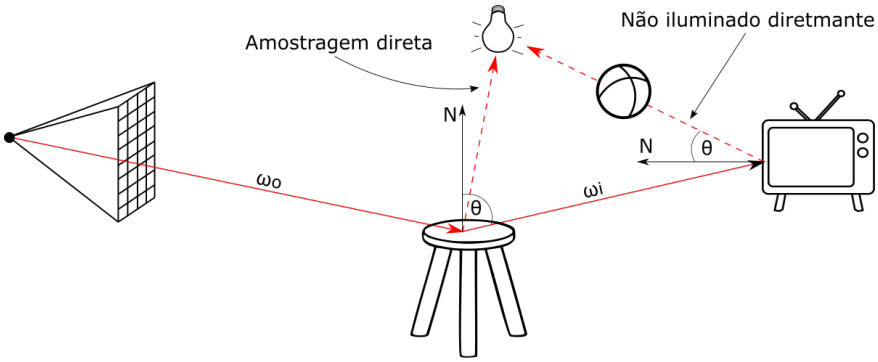


Figura 13: Amostragem direta da luz.

Sabe-se que o termo $L_i(x', \omega_i)$ da equação 4, pode ser escrito como a soma entre a radiância emitida por um superfície x' na direção ω_i e a radiância espalhada por ela nesta mesma direção (SHIRLEY; WANG; ZIMMERMAN, 1996):

$$L_i(x', \omega_i) = L_e(x', \omega_i) + L_s(x', \omega_i). \tag{16}$$

Substituindo o termo $L_i(x', \omega_i)$ da equação 4 pela equação 16 e reescrevendo como uma soma de integrais, temos que:

$$L_s(x, \omega_o) = \int_{\Omega_L} L_e(x', \omega_i)\ f_s(x, \omega_o, \omega_i)\ \ |\cos\theta_i|\ \ d\omega_i + \int_{\Omega_T} L_s(x', \omega_i)\ f_s(x, \omega_o, \omega_i)\ \ |\cos\theta_i|\ \ d\omega_i,$$

onde Ω_L é o ângulo sólido da luz, como mostrado na figura 14. A integral do lado esquerdo é a equação que representa a iluminação direta da luz, que sai de alguma fonte de luz x' e incide em x pela direção ω_i:

$$L_d(x, \omega_o) = \int_{\Omega_L} L_e(x', \omega_i)\ f_s(x, \omega_o, \omega_i)\ \ |\cos\theta_i|\ \ d\omega_i.$$

Desta forma, a equação de *rendering* pode ser escrita como uma soma de duas integrais: uma cujo o domínio de integração é o ângulo sólido da luz Ω_L e outra cujo domínio de integração é restante do hemisfério Ω_T:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + L_d(x, \omega_o) + \int_{\Omega_T} L_s(x', \omega_i)\ f_s(x, \omega_o, \omega_i)\ \ |\cos\theta_i|\ \ d\omega_i, \tag{17}$$

onde $\Omega_T = \Omega - \Omega_L$.

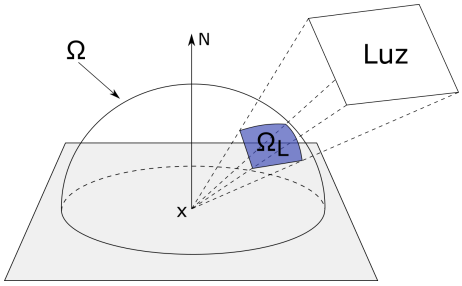


Figura 14: Ângulo sólido da luz.

Adamsen (2009) apresenta a equação de *rendering* com amostragem direta da luz, formulada no espaço dos caminhos e estimada com o método de Monte Carlo:

$$L_p = \frac{1}{N} \sum_{i=1}^{N} L_{pi}, \tag{18}$$

onde L_p é a radiância estimada para um *pixel* tomando uma média de N amostras, e L_{pi} é a radiância estimada de uma única amostra, escrita da forma:

$$L_{pi} = \sum_{s=0}^{N_s} C(s), \tag{19}$$

onde N_S é o tamanho máximo permitido para um caminho, s é o tamanho do caminho a cada iteração e $C(s)$ é a contribuição de um único caminho de tamanho s, escrita como:

$$\begin{aligned} C(s) = \frac{L_e(y_0, \overrightarrow{y_0 x_s})}{p(y_0)} \; f_s(x_s, \overrightarrow{x_s y_0}, \overrightarrow{x_s x_{s-1}}) \; G(y_0 \leftrightarrow x_s) \\ \left(\prod_{i=1}^{s-1} \frac{f_s(x_i, \overrightarrow{x_i x_{i+1}}, \overrightarrow{x_i x_{i-1}}) \; |N_{x_i} \cdot \overrightarrow{x_i x_{i+1}}|}{p(\overrightarrow{x_i x_{i+1}})} \right), \end{aligned} \tag{20}$$

onde y_0 é um ponto da superfície da luz, $L_e(y_0, \overrightarrow{y_0 x_s})$ é a radiância emitida a partir de y_0 em direção a um determinado ponto x_s, e $p(y_0)$ é a probabilidade de escolher aleatoriamente o ponto y_0 em uma superfície de luz com área A, dado por:

$$p(y_0) = \frac{1}{A}.$$

2.6 Path Tracing Bidirecional

O algoritmo de *path tracing*, mesmo utilizando amostragem direta da luz, ainda possui problemas em situações onde a maior parte da geometria da cena não é iluminada diretamente. Quando isso acontece, a maior parte dos testes de visibilidade entre as superfícies e a luz irá falhar, desperdiçando tempo e processamento. Neste contexto, o *path tracing* bidirecional surge com o objetivo de melhorar o desempenho nestas situações.

O *path tracing* bidirecional foi proposto por Lafortune e Willems (1993). A ideia do algoritmo é construir dois caminhos que percorrem uma cena: um caminho cujo raio inicial parte da superfície da luz, e um caminho cujo raio inicial parte da câmera. Estes caminhos são representados pelos pontos das superfícies em que a luz interagiu durante a sua trajetória. Uma vez obtidos esses caminhos, o algoritmo tenta "conectar" os pontos entre eles, com o objetivo de formar um novo caminho. Se dois vértices dos caminhos puderem ser conectados, isto significa que o novo caminho formado é um caminho válido entre a câmera e a fonte de luz.

A figura 15 mostra o exemplo de dois caminhos formados após as interação dos raios com as superfícies. O caminho que inicia da câmera (setas vermelhas), possui os pontos x_0, x_1 e x_2, enquanto o caminho que inicia da luz (seta azul), possui os pontos y_0 e y_1. As linhas tracejadas representam as conexões que podem ser feitas para formar um novo caminho.

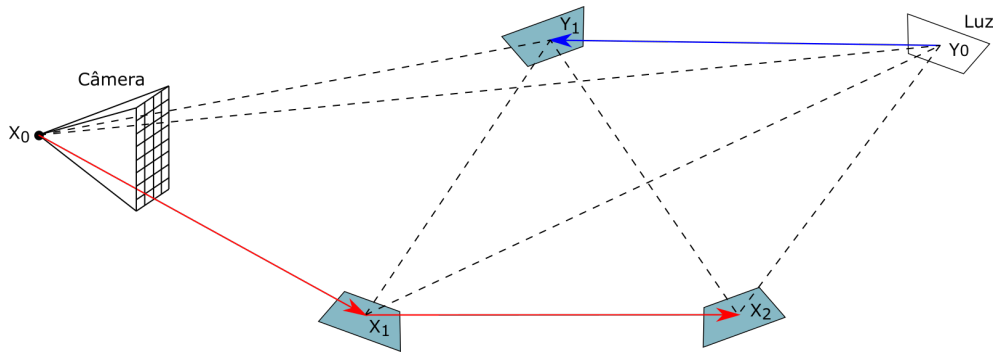


Figura 15: Exemplo de caminhos formados.

Adamsen (2009) apresenta a equação de *rendering* para o método bidirecional, formulada no espaço dos caminhos e utilizando o método de integração de Monte Carlo:

$$L_p = \frac{1}{N} \sum_{i=0}^{N} L_{pi}, \tag{21}$$

onde L_p é a radiância estimada para um *pixel* tomando uma média de N amostras, e L_{pi} é a radiância estimada de uma única amostra, escrita da forma:

$$L_{pi} = \sum_{s=0}^{N_S} \sum_{t=0}^{N_T} W(s,t)\, C(s,t), \tag{22}$$

onde N_S e N_T são respectivamente os tamanhos máximos definidos para os caminhos da câmera e da luz. $C(s,t)$ é a função que mede a contribuição do novo caminho formado pela conexão dos vértices x_s e y_t. A função $W(s,t)$ é uma função que mede o peso de cada contribuição $C(s,t)$. A regra geral para a função $W(s,t)$ é que o somatório dos pesos de caminhos de mesmo comprimento deve ser igual a 1.

Existem quatro casos distintos para os quais deve-se avaliar a função $C(s,t)$:

- Quando $s = 0$ e $t = 0$: Isto significa tentar conectar a câmera diretamente a um ponto da luz. Nesta caso, a função pode ser calculada da forma:

$$C(0,0) = L_e(y_0, \overrightarrow{y_0 x_0}) G(y_0 \leftrightarrow x_0);$$

- Quando $s > 0$ e $t = 0$: Esta situação corresponde exatamente ao *path tracing* com amostragem direta da luz:

$$\begin{aligned} C(s,0) = \frac{L_e(y_0, \overrightarrow{y_0 x_s})}{p(y_0)} \; & f_s(x_s, \overrightarrow{x_s y_0}, \overrightarrow{x_s x_{s-1}}) \; G(y_0 \leftrightarrow x_s) \\ & \left(\prod_{i=1}^{s-1} \frac{f_s(x_i, \overrightarrow{x_i x_{i+1}}, \overrightarrow{x_i x_{i-1}}) \; |N_{x_i} \cdot \overrightarrow{x_i x_{i+1}}|}{p(\overrightarrow{x_i x_{i+1}})} \right); \end{aligned} \tag{23}$$

- Quando $s = 0$ e $t > 0$, neste caso a função fica da forma:

$$\begin{aligned} C(0,t) = \frac{L_e(y_0, \overrightarrow{y_0 y_1})}{p(y_0, \overrightarrow{y_0 y_1})} \; & f_s(y_t, \overrightarrow{y_t y_{t-1}}, \overrightarrow{y_t x_0}) \; G(y_t \leftrightarrow x_0) \\ & \left(\prod_{j=1}^{t-1} \frac{f_s(y_j, \overrightarrow{y_j y_{j-1}}, \overrightarrow{y_j y_{j+1}}) \; \left|N_{y_j} \cdot \overrightarrow{y_j y_{j+1}}\right|}{p(\overrightarrow{y_j y_{j+1}})} \right); \end{aligned} \tag{24}$$

- Por fim, quando $s > 0$ e $t > 0$: Este é o caso geral, quando existe a conexão de pontos intermediários dos caminhos da câmera e da luz:

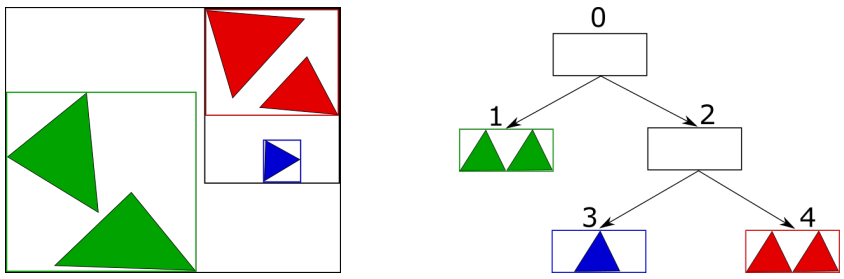
$$\begin{aligned} C(s,t) = \frac{L_e(y_0, \overrightarrow{y_0 y_1})}{p(y_0, \overrightarrow{y_0 y_1})} \; & \left(\prod_{j=1}^{t-1} \frac{f_s(y_j, \overrightarrow{y_j y_{j-1}}, \overrightarrow{y_j y_{j+1}}) \; \left|N_{y_j} \cdot \overrightarrow{y_j y_{j+1}}\right|}{p(\overrightarrow{y_j y_{j+1}})} \right) \\ & f_s(y_t, \overrightarrow{y_t y_{t-1}}, \overrightarrow{y_t x_s}) \; G(y_t \leftrightarrow x_s) \; f_s(x_s, \overrightarrow{x_s y_t}, \overrightarrow{x_s x_{s-1}}) \\ & \left(\prod_{i=1}^{s-1} \frac{f_s(x_i, \overrightarrow{x_i x_{i+1}}, \overrightarrow{x_i x_{i-1}}) \; |N_{x_i} \cdot \overrightarrow{x_i x_{i+1}}|}{p(\overrightarrow{x_i x_{i+1}})} \right). \end{aligned} \tag{25}$$

2.7 Estruturas de Aceleração

As malhas das cenas 3D são representadas por faces poligonais. O polígono mais simples e utilizado é o triângulo (AZEVEDO; CONCI, 2003). Deste modo, para determinar os pontos dos caminhos que a luz percorre na cena, é preciso realizar testes de interseção entre os raios de luz e os triângulos que compõem a cena. Um algoritmo de percorrimento "força bruta" precisa realizar testes com todos os triângulos da cena para determinar qual deles está na direção do raio de luz. Uma consequência desta abordagem é que o número de testes aumenta de forma proporcional ao número de triângulos da cena, ou seja, um algoritmo com complexidade de pior caso $O(n)$.

As estruturas de aceleração possuem como objetivo diminuir a complexidade dos algoritmos de percorrimento e consequentemente o número de testes que precisão ser realizados. Muitas estruturas, no contexto de renderização de cenas 3D, já são estudadas e utilizadas há algumas décadas. Algumas das estruturas mais conhecidas são: *BSP-tree* (IZE; WALD; PARKER, 2008), *kd-tree* (RESHETOV; SOUPIKOV; HURLEY, 2005) e Grid (WALD et al., 2006). A estrutura que mais vem sendo utilizada em conjunto com algoritmos de renderização de cenas 3D é a BVH (*bounding volume hierarchy*), pois ela demostrou ser bastante eficiente tanto para renderização de cenas estáticas, quanto para cenas dinâmicas (WALD; BOULOS; SHIRLEY, 2007).

A BVH consiste em uma estrutura hierárquica no formato de árvore. Em seu algoritmo de construção os objetos da cena são subdivididos em espaços chamados "volumes delimitadores". Cada objeto, triângulo ou conjunto deles, são delimitados por estes volumes. A hierarquia dessa estrutura é construída de forma que o volume que delimita toda a cena fique armazenado no nó raiz da árvore, seus nós internos armazenam volumes menores, e os nós folha armazenam volumes que delimitam um ou mais triângulos. A figura 16a mostra um exemplo de um espaço com vários triângulos e seus volumes delimitadores. A figura 16b mostra a hierarquia formada durante a construção da BVH.



(a) (b)

Figura 16: BVH. (a) Triângulos e volumes delimitadores, (b) A hierarquia.

A ideia de utilizar uma estrutura do tipo árvore, consiste em realizar testes de intersecção entre os raios e os volumes delimitadores de cada nó. Se um raio não atravessa o volume de um nó, então não é necessário realizar testes com toda a sub-árvore formada por ele, diminuindo assim o número de testes que precisam ser realizados. Supondo que um percorrimento seja realizado em apenas um ramo da árvore e o algoritmo identifique o triângulo mais próximo na direção do raio, o algoritmo teria realizado cerca de $log(n)$ testes de intersecção. Infelizmente, as árvore nem sempre são balanceadas e as heurísticas utilizadas na construção geralmente geram nós cujos filhos possuem volumes com intersecções entre si (WALD; BOULOS; SHIRLEY, 2007). Sendo assim, os algoritmos precisam percorrer vários ramos da árvore para poder descobrir o triângulo mais próximo. Ainda assim, os algoritmos que utilizam estas estruturas realizam um número significativamente menor de testes quando comparados aos que utilizam força bruta.

3 METODOLOGIA

Esta seção descreve as etapas que serão realizadas no desenvolvimento da aplicação, as escolhas algorítmicas realizadas e os testes que serão aplicados. Inicialmente será apresentado o ambiente em que a aplicação será desenvolvida, as linguagens de programação escolhidas e as ferramentas utilizadas. Em seguida, serão apresentadas as etapas de desenvolvimento e as escolhas algorítmicas. Por fim, serão apresentadas as metodologias utilizadas para os testes.

3.1 Ambiente de Desenvolvimento

A aplicação será desenvolvida, compilada e executada utilizando os sistemas operacionais Windows 10 e Ubuntu 16.04. O computador utilizado possui 8 GB de RAM, processador Intel Core i5 com frequência 3.2 GHz e processador gráfico Intel HD *Graphics* 4000. As linguagens de programação utilizadas para o desenvolvimento do são: C++11, GLSL e OpenCL-C (versão 1.2).

3.1.1 Ferramentas Utilizadas

Algumas ferramentas são essenciais para o desenvolvimento da aplicação, de modo que sem elas, a aplicação não poderá ser desenvolvida em tempo hábil. As ferramentas que serão utilizadas são apresentadas a seguir:

- **Assimp:** Biblioteca que fornece funções e estruturas que são utilizadas para o carregamento das cenas 3D;
- **SDL2:** Biblioteca utilizada para criar e gerenciar a janela da aplicação, na qual serão exibidos o resultados da renderização progressiva.
- **SDL Image:** Biblioteca que fornece funções para o carregamento de imagens. Neste trabalho ela é utilizada para carregar as texturas das cenas 3D.
- **OpenCL:** API utilizada para escrever programas que funcionam em plataformas heterogêneas, como CPUs e GPUs. Neste trabalho ela é utilizada para escrever executar os algoritmos de *path tracing* em GPU.
- **OpenGL:** API que fornece mecanismos e funções pra o desenvolvimento de aplicações gráficas. Neste trabalho ela é utilizada para rasterizar na janela da aplicação as imagens geradas pelo algoritmo de *path tracing*.

3.2 Etapas de Desenvolvimento

O desenvolvimento do trabalho será realizado em duas etapas. A primeira etapa consiste na implementação do algoritmo de *path tracing* original e do algoritmo que utiliza amostragem direta da luz. Na segunda etapa, será implementado o algoritmo de *path tracing* bidirecional, que deve reutilizar parte do que foi implementado na etapa anterior.

Os algoritmos de *path tracing* serão executados em GPU, cuja programação deve ser realizada utilizando a API OpenCL. A ideia é programar uma aplicação em CPU que trabalha gerenciando as atividades executadas em GPU. Por sua vez, os algoritmos executados em GPU deverão calcular e atualizar os valores dos *pixels* da imagem que deve ser renderizada. Esta imagem consiste em um *buffer* que será compartilhado com a API OpenGL, que por sua vez irá rasterizá-la na tela.

Na etapa de desenvolvimento dos algoritmos, será construída e utilizada uma estrutura de aceleração, com o objetivo de tornar a renderização mais rápida. Raios primários serão construídos e utilizados como ponto de partida para determinar os caminhos da luz. Estes raios deverão percorrer a cena através da estrutura de aceleração e determinar os pontos de intersecção com os objetos, e a partir deles, construir raios secundários. Para determinar os pontos de intersecção, será utilizado um algoritmo de interseção raio-triângulo, uma vez que a geometria da cena consiste em malhas formadas por triângulos. Durante a construção dos caminhos, pretende-se armazenar todos os dados que são necessários para calcular a equação de *rendering*, dados como: ângulos de incidência e de saída, tipo de material, vetores normais, BSDFs, PDFs, entre outros. Uma vez obtidos todos dados, a equação de *rendering* será calculada utilizando o método de Monte Carlo.

3.3 Escolhas Algorítmicas

A estrutura de aceleração escolhida foi a BVH, pois ela demonstra ser uma das mais eficientes, tanto para a renderização de cenas estáticas, quanto para dinâmicas (WALD; BOULOS; SHIRLEY, 2007). Neste trabalho não serão utilizadas cenas dinâmicas, porém o objetivo é contruí-lo de maneira que possa ser expandido no futuro. A BVH será implementada utilizando o método de construção SAH (*Surface Area Heuristic*), como é demonstrado por Wald, Boulos e Shirley (2007). A estrutura de volume delimitador escolhida foi a AABB (*Axis-Aligned Bounding Box*), pelo fato de ser simples e exigir poucas informações para o seu armazenamento. Neste caso, será preciso implementar um algoritmo de interseção raio-AABB, para que a estrutura da BVH possa ser percorrida. Para este fim, será utilizado o algoritmo proposto por Aila, Laine e Karras (2012), que além de ser relativamente recente, demonstra ser bastante eficiente para percorrimento de cenas em GPU.

Para o algoritmo de intersecção raio-triângulo, o escolhido foi o proposto por Möller e Trumbore (1997). O motivo da escolha foi o fato de que o algoritmo funciona considerando uma quantidade de dados mínima (referentes ao triângulo). Desta forma, apenas uma pequena quantidade de dados precisam ser lidos da memória de vídeo no momento da execução dos algoritmos.

3.4 Metodologia de Testes

Após as implementações serem realizadas, testes precisam ser realizados para validá-las. Os testes serão realizados levando em consideração tanto o desempenho, quanto a qualidade dos algoritmos. Inicialmente será realizada uma comparação entre os resultados obtidos neste trabalho e os resultados de outros *softwares* que já são reconhecidos pela comunidade da área. Por fim, serão realizados testes para verificar e comparar os desempenhos dos algoritmos de *path tracing* implementados.

3.4.1 Testes de Qualidade dos Algoritmos

O *software* escolhido para comparar seus resultados com os obtidos neste trabalho foi o *Mitsuba Renderer* (JAKOB, 2010). Para comparar os resultados, serão renderizadas duas imagens da mesma cena, uma em cada *software*. As imagens precisão representar exatamente a mesma região de uma cena e possuírem a mesma resolução. Uma vez obtidas as duas imagens, os *pixels* correspondentes entre elas serão comparados para determinar as diferenças. Ao final, deve-se obter uma terceira imagem que representa as diferença entre as duas originais. Para calcular a diferença entre os *pixels* de cada imagem, primeiro será necessário calcular a média das componentes RGB de cada *pixel* da seguinte forma:

$$M = \frac{R + G + B}{3},$$

onde M é a média e R, G e B são os valores das componentes dos *pixels*. Uma vez obtidas as médias dos *pixels* das duas imagens, os *pixels* da terceira imagem serão calculados como sendo a diferença entre eles:

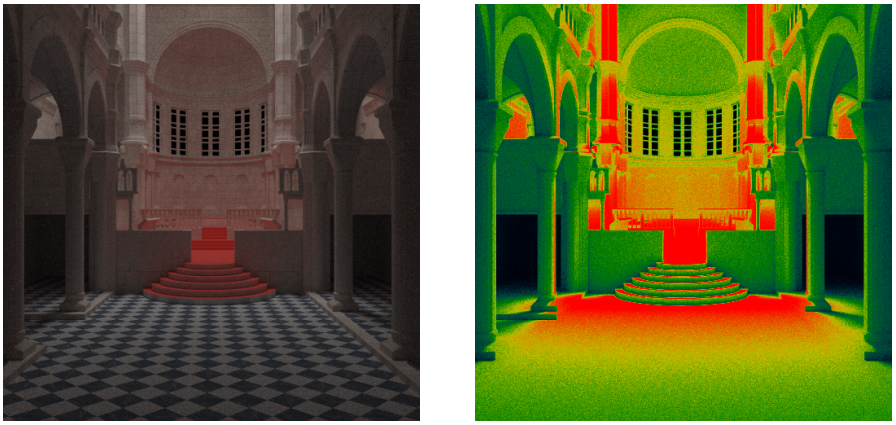
$$D = |M_a - M_b| .$$

onde D é a diferença, M_a e M_b são as médias dos *pixels* equivalentes de cada imagem.

3.4.2 Testes de Eficiência dos Algoritmos

Para testar a eficiência dos algoritmos de *path tracing*, serão realizadas comparações entre três variações do algoritmo: o algoritmo original, o algoritmo que utiliza amostragem direta da luz e o algoritmo bidirecional. O objetivo é testar a eficiência dos algoritmos ao

"encontrar" a luz, em outras palavras, verificar qual deles é mais eficiente ao encontrar caminhos entre a câmera e a fonte de luz. Para medir a eficiência, este trabalho propõe a utilização de imagens no formato HSV, também conhecidas como "mapas de calor". Como não foi encontrado na literatura um modo de visualização semelhante (neste contexto), este modo de visualização trata-se de uma contribuição deste trabalho. Um exemplo de mapa de calor pode ser visto na figura 17b.



(a) (b)

Figura 17: Exemplo de mapa de calor. (a) Imagem original, (B) HSV.

Os mapas de calor serão gerados da seguinte forma: será renderizada uma imagem onde o algoritmo em questão é executado N vezes para cada *pixel*. Para cada execução, será verificado se o algoritmo conseguiu ou não determinar um caminho entre a câmera e a fonte de luz. Ao final, cada *pixel* terá um valor N_s de sucessos onde um caminho foi encontrado, de modo que $N_s \leq N$. O percentual de eficiência (P) de cada *pixel* será calculado da seguinte forma:

$$P = \frac{N_s}{N} \cdot 100\%.$$

Uma vez determinado o percentual de eficiência de cada *pixel*, o mapa de calor será construído utilizando estes valores, onde cada *pixel* do mapa irá receber uma cor de acordo com o seu valor, conforme mostrado na figura 18.

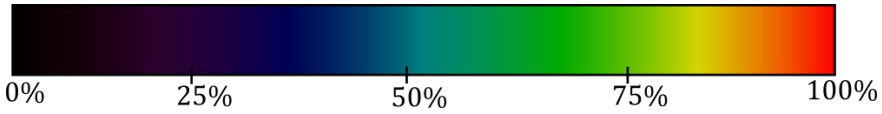


Figura 18: Relação entre o percentual de eficiência e a cor do *pixel*.

O mapa de calor funciona de seguinte forma: quanto mais vermelha for uma região da cena, significa que o algoritmo utilizado foi mais eficiente ao encontrar a luz nesta região. Por outro lado, quanto mais escura for a região, significa que o algoritmo foi menos eficiente ao encontrar a luz nesta região.

Além da eficiência ao encontrar a luz, também serão verificados os tempos em que cada algoritmo gasta para completar uma execução. Esta medida servirá para determinar o custo benefício de cada algoritmo, pois de nada adiantaria um algoritmo que encontra a luz com $2x$ mais facilidade, se ele gasta $4x$ mais tempo para ser executado. Para a obtenção dos tempos, cada algoritmo será executados N vezes, onde o tempo de cada execução será computado e valor de tempo final irá consistir na média dos tempos de cada uma das N execuções.

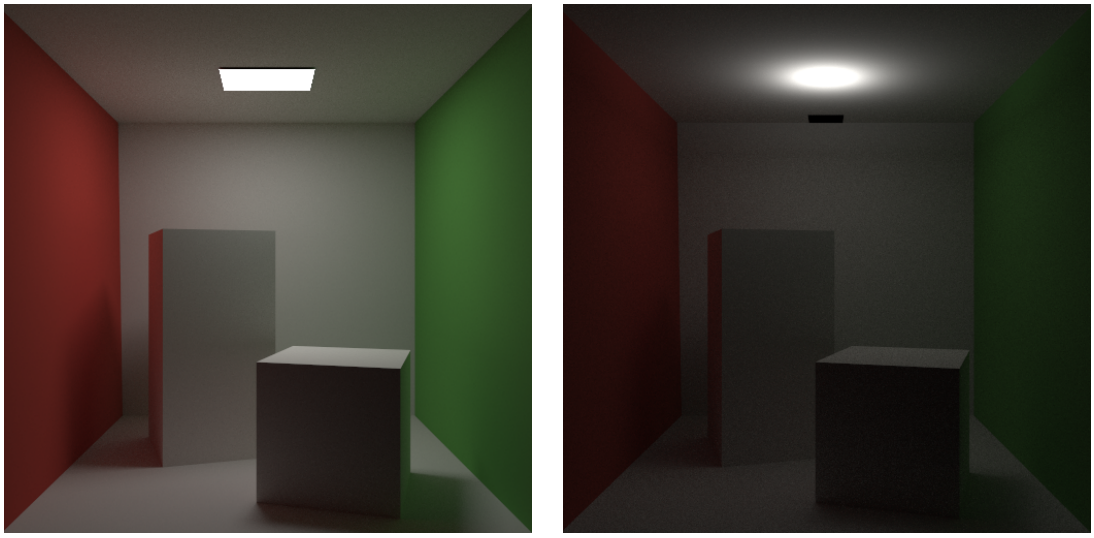
3.5 Cenas de Testes

As cenas de testes foram escolhidas levando em consideração os testes que serão realizados. Os testes serão realizados em um total de 6 cenas, sendo 3 cenas obtidas da base de dados fornecida por McGuire (2017) e três cenas que consistem em variações da *Cornell Box* (GORAL et al., 1984), onde as posições ou tamanhos das fontes de luz são modificados de modo a permitir testes de situações específicas. As cenas escolhidas para os testes podem ser vistas na tabela 1 e na figura 19.

Tabela 1: Cenas de testes.

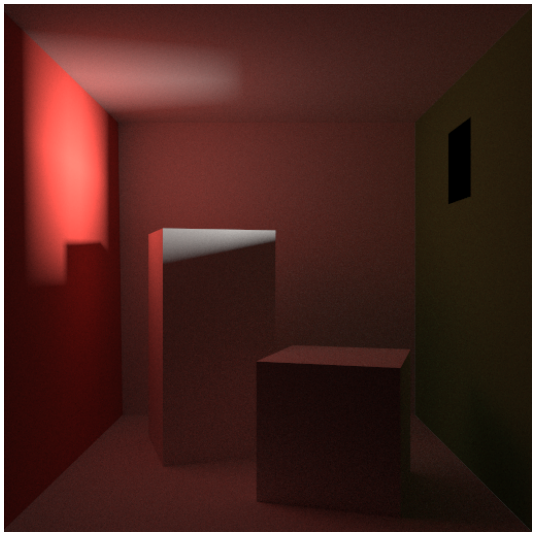
Cena	Nome	Nº de Triângulos	Repositório
1	*Cornell Box* 1	32	(JAKOB, 2010) Editado
2	*Cornell Box* 2	32	(JAKOB, 2010) Editado
3	*Cornell Box* 3	40	(JAKOB, 2010) Editado
4	*Crytek Sponza 1*	200.431	(MCGUIRE, 2017) Editado
5	*Crytek Sponza 2*	200.431	(MCGUIRE, 2017) Editado
6	*Sibenik Cathedral*	75.285	(MCGUIRE, 2017) Editado

A cena 1 (figura 19a) consiste na clássica cena da *Cornell Box* desenvolvida por Goral et al. (1984) e amplamente utilizada pela comunidade da área para testar algoritmos de renderização. A cena 2 (figura 19b) é uma variação da cena 1, onde a área da fonte de luz foi diminuída e apontada para o teto. Esta modificação foi feita com o objetivo de fazer com que a maior parte da cena não seja iluminada diretamente. A cena 3 (figura 19c) também consiste em uma variação de cena 1, porém neste caso, a fonte de luz foi colocada fora da *Cornell Box*, de modo que a luz só pode passar através de um orifício no topo da parede direita. A cena 4 (figura 19d) consiste no modelo 3D do Palácio de Sponza (Croácia), desenvolvido pela Crytek. Um fonte de luz com uma área muito grande foi colocada no topo do palácio (fora), de modo a parecer que ele está sendo iluminado pela luz do sol. A cena 5 (figura 19e) consiste em uma variação da cena 4, onde foi colocada apenas uma pequena fonte de luz ao final do corredor. A cena 6 (figura 19f) consiste no modelo 3D da Catedral de São Tiago (Croácia), desenvolvido por Marko Dabrovic. Uma fonte de luz foi colocada no topo da torre da catedral, de modo que o altar receba a maior parte da iluminação direta.



(a)

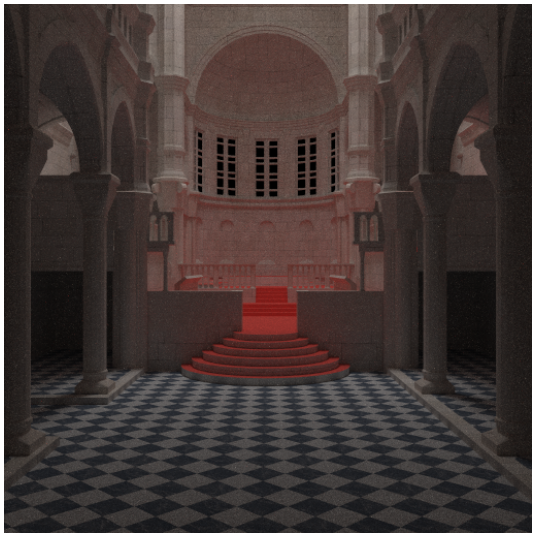
(b)



(c)

(d)

(e)



(f)

Figura 19: Cenas de teste. (a) *Cornell Box* 1, (b) *Cornell Box* 2, (c) *Cornell Box* 3, (d) *Crytek Sponza 1*, (e) *Crytek Sponza 2*, (f) *Sibenik Cathedral*.

4 DESENVOLVIMENTO DO TRABALHO

Esta seção apresenta as etapas realizadas no desenvolvimento da aplicação. Inicialmente será apresentada a estrutura geral da aplicação desenvolvida. Em seguida, serão apresentados os procedimentos de construção e linearização da BVH, que é necessário para que ela possa ser utilizada em conjunto com os algoritmos executados em GPU. Por fim, serão apresentados os desenvolvimentos dos algoritmos de *path tracing* com o método de iluminação direta da luz e bidirecional.

O desenvolvimento da aplicação é dividido em duas partes principais: a programação do *host* e programação dos *kernels*. Os *kernels* são programas escritos em OpenCL-C, e neles ficam os algoritmos de *path trancing*. O *host* é um dispositivo que controla a execução dos *kernels* através da API OpenCL. Neste projeto, o *host* é uma CPU, que é programada utilizando a linguagem de programação C++.

4.1 Estrutura do Host

O programa executado pelo *host* é responsável por gerenciar a aplicação como um todo. As tarefas do *host* consistem em carregar as cenas 3D, construir as estruturas de aceleração, carregar os dados necessários da RAM para a VRAM, além de controlar a execução dos *kernels* em GPU.

Em seu fluxo de execução, o *host* inicia criando a janela de exibição, em seguida carrega a malha da cena 3D, constrói a BVH, os *buffers*, e carrega na VRAM os dados necessários para a execução dos *kernels*. Por fim, o *host* carrega o *kernels* a partir do disco e inicia o *loop* de renderização. O *loop* de renderização por sua vez, realiza a chamada para a execução de cada *kernel*, calcula a média das amostras e atualiza a imagem na tela. Os principais passos do fluxo de execução do *host* podem ser vistos na figura 20.

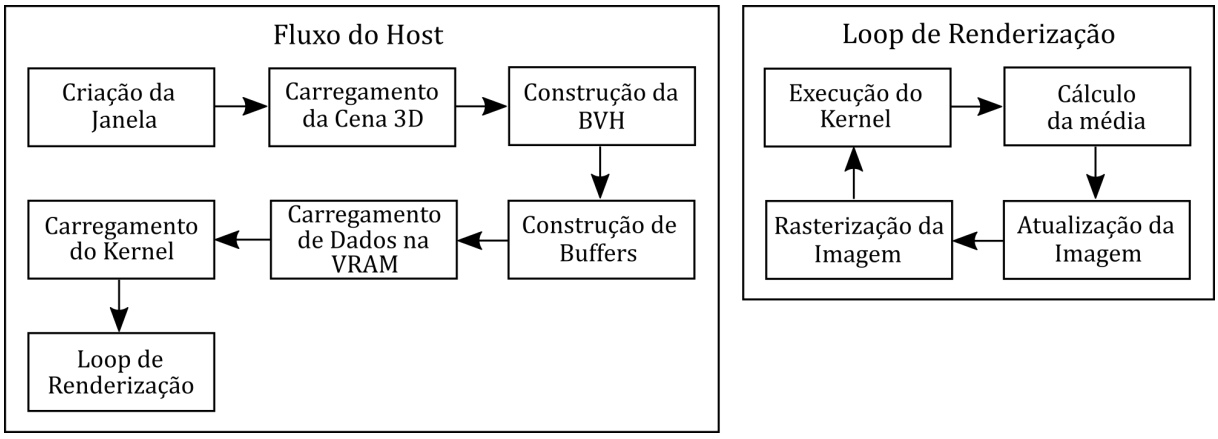


Figura 20: Fluxo de execução do *host*.

4.1.1 Classes e Estruturas Utilizadas

Aqui serão apresentadas a principais classes e estruturas utilizadas pelo *host*. Além dessas estruturas, também serão apresentados os *buffers* que o *host* precisa carregar na VRAM para que os *kernels* possam utilizá-los durante a execução dos algoritmos. As principais estruturas são listadas a seguir:

- **Estrutura da Câmera:** Possui dados referentes a posição da câmera, a direção para onde ela olha, além de pontos e vetores que serão utilizados nos *kernels* para construir os raios primários;
- **Estrutura do Triângulo:** Possui os dados dos vértices que formam um triângulo, como a suas posições, seus vetores normais e suas coordenadas de textura. Também possui dados relacionados ao triângulo em geral, como a sua cor, o tipo de material, índices de refração e rugosidade, entre outros;
- **Estrutura da BVH:** Possui os nós da BVH. Cada nó possui dados referentes ao volume que ele delimita na cena, além de ponteiros para os nós filhos. Os volumes que os nós delimitam são representados pela estrutura da AABB. A AABB possui dados referentes aos pontos máximos e mínimos do volume delimitado.

A partir destas estruturas e de outros tipos de dados, *buffers* são criados e carregados na VRAM através da API OpenCL. O principais *buffers* que são utilizados nos *kernels* são listados a seguir:

- ***Buffer* da Câmera**. *Buffer* que possui a estrutura da Câmera.
- ***Buffer* de Triângulos**. Vetor de estruturas do tipo Triângulo, que possui todos os triângulos da cena. É utilizado nos *kernels* para realizar testes de interseção entre raios e os triângulos da cena.
- ***Buffer* da BVH**. *Buffer* que possui a estrutura da BVH linearizada, em outras palavras, a estrutura da BVH no formato de um vetor. Este *Buffer* é utilizado nos *kernels* para realizar o percorrimento da cena. Detalhes sobre o processo de linearização serão apresentados na subseção 4.3.
- ***Buffer* de *Seeds***. *Buffer* que contém *seeds* (sementes) que são utilizados nos *kernels* em funções geradoras de números pseudo-aleatórios.
- ***Color Buffer***. *Buffer* no formato de imagem 2D que é compartilhado entre os *kernels* e o *host*. O *kernel* preenche este *buffer* com os valores das amostras de radiância. O *host* utiliza este *buffer* para rasterizar a imagem na tela através da API OpenGL.

4.2 Estrutura dos Kernels

Neste trabalho foram desenvolvidos três *kernels*, cada um com uma variação do algoritmo de *path tracing*: um com o algoritmo original, um com o algoritmo que realiza amostragem direta da luz, e um com o algoritmo bidirecional. O trabalho destes *kernels* consistem de forma resumida em: gerar os raios primários, executar o algoritmo de *path tracing*, e ao final, escrever o valor da amostra de radiância no *color buffer*. Os passos do fluxo de execução dos *kernels* podem ser visualizados na figura 21.

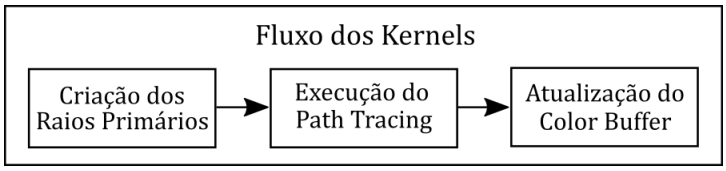


Figura 21: Fluxo de execução dos *kernels*.

Os *kernels* possuem as mesmas estruturas que o *host*, como as estruturas do Triângulo, da Câmera e da BVH. Adicionalmente, os *kernels* possuem a estrutura do Raio, que possui dados referentes a origem e direção dos raios de luz.

Para cada *pixel* da imagem que será renderizada, é criada uma *thread* que possui uma cópia do *kernel*. Durante sua execução, cada *thread* fica responsável por criar o raio primário, executar o algoritmo de *path tracing* e atualizar o *color buffer* somente no *pixel* pelo qual ela é responsável. A linguagem de programação OpenCL-C fornece mecanismos para identificar cada *thread* através de números identificadores (ID's). No caso deste trabalho, como pretende-se executar *kernels* bidimensionais para renderizar imagens, cada *thread* possuis dois ID's. Estes ID's são utilizados pelas *threads* para escrever o valor da amostra na posição correta do *color buffer*.

4.3 Construção e Linearização da BVH

No programa do *host*, a BVH foi implementada com base no método proposto por Wald, Boulos e Shirley (2007). Nesta implementação, a estrutura da BVH consiste em uma árvore binária, onde cada nó da árvore possui dados referentes a AABB, os ponteiros para os nós filhos e a lista de triângulos delimitados pela AABB (caso seja um nó folha).

Infelizmente, na programação dos *kernels* não é possível utilizar a estrutura do modo como ela foi construída no *host*, pois a API OpenCL não fornece mecanismos para alocação dinâmica e manipulação de ponteiros na VRAM. Neste caso, a única solução possível é linearizar a estrutura, em outras palavras, converter a estrutura de árvore em um vetor de nós, de modo que não seja necessário a utilização de ponteiros.

Para construir a árvore linearizada, primeiro é preciso definir uma nova estrutura para o nós, pois eles não poderão mais possuir ponteiros para os nós filhos. A solução encontrada foi substituir os ponteiros por números inteiros que indicam quais são as posições (índices) dos seus filhos no vetor de nós. A nova estrutura do nó possui então os dados da AABB e dois campos (E e D), que são os números inteiros que indicam as posições dos filhos. Uma ilustração da nova estrutura do nó é mostrada na figura 22.

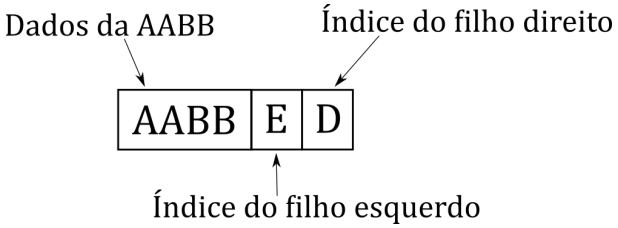


Figura 22: Estrutura do nó da BVH linearizada.

Com a estrutura do nó criada até o momento, ainda restam dois problemas a serem solucionados: como identificar um nó folha e como identificar os triângulos que a AABB de um nó folha delimita. Em uma implementação que utiliza ponteiros, um nó folha é identificado quando os valores dos ponteiros de seus dois nós filhos são nulos. Devido a limitações descritas anteriormente, não é possível utilizar esta abordagem no programa dos *kernels*. Já para identificar os triângulos que um nó folha delimita, bastaria adicionar a lista destes triângulos na estrutura do nó. Porém, a solução utilizada neste trabalho consegue resolver estes dois problemas sem adicionar mais dados na estrutura do nó.

Na solução encontrada, um nó folha é identificado quando os campos E e D de sua estrutura possuírem valores menores ou iguais a zero. Quando isto ocorrer, para obter os triângulos que a AABB deste nó delimita, basta multiplicar os valores de E e D por -1, o resultado destas operações irá indicar um intervalo no vetor de triângulos, cujos triângulos contidos nele são os delimitados pela AABB. Uma representação visual desta solução é mostrada na figura 23.

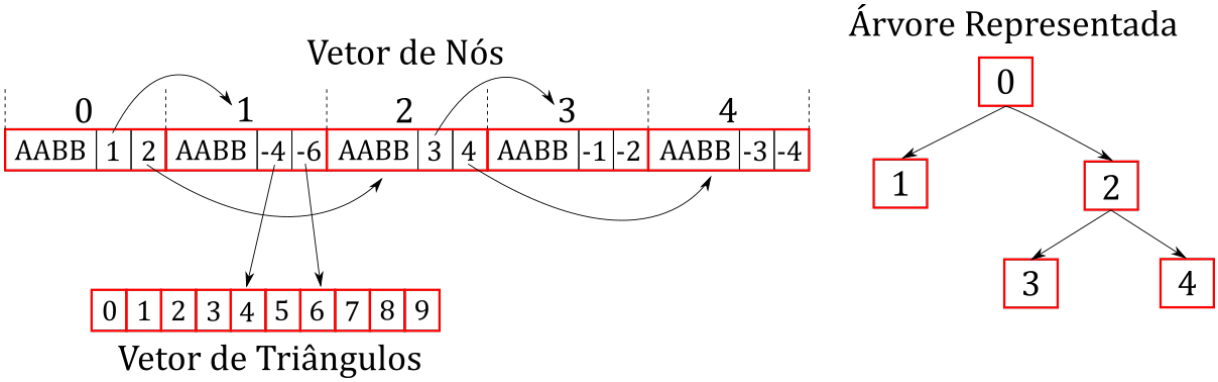


Figura 23: Representação da BVH como um vetor de nós.

4.4 Construção dos Raios Primários

A construção dos raios primários consiste em determinar a origem e a direção dos raios que saem da câmera. A origem e direção desses raios são representados pelos símbolos O e $\vec{D}$, respectivamente. Determinar a origem deses raios é simples, pois consiste exatamente no mesmo valor da posição da câmera. Porém, para definir a direção $\vec{D}$, é preciso levar em consideração o *pixel* que cada *thread* é responsável por calcular a cor. Os ID's das *threads* são utilizados para identificar esses *pixels*, e a partir desta informação, o raio primário pode ser construído com a direção correta.

Para determinar a direção do raio primário, é necessário primeiro definir a região do espaço que representa o sensor do *pixel*. Para definir essa região, é necessário calcular o seu ponto superior esquerdo. O cálculo deste ponto é realizado utilizando os ID's das *threads* e os dados fornecidos pela estrutura da câmera:

$$P_{xy} = P_i + \vec{P_w} \cdot x + \vec{P_h} \cdot y,$$

onde P_{xy} é o ponto superior esquerdo do *pixel* com coordenadas (x, y). P_i é ponto superior esquerdo de toda a imagem. Os valores x e y são os ID's das *threads* e também as coordenadas do *pixel* na imagem. Por fim, $\vec{P_w}$ e $\vec{P_h}$ são vetores de deslocamento que indicam para que direção os *pixels* da imagem se distribuem (ver figura 24).

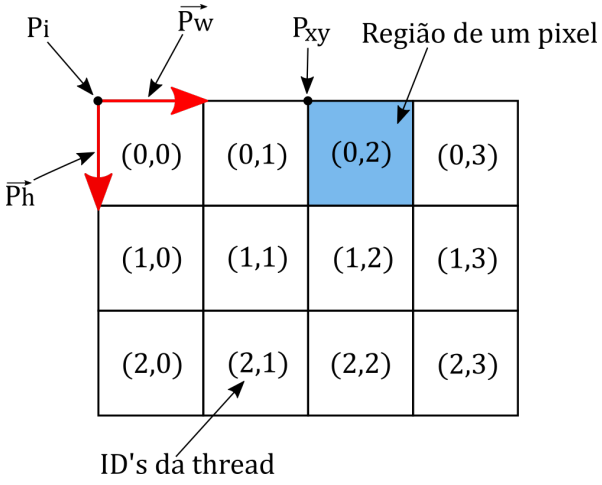


Figura 24: Dados utilizados para determinar a região de um *pixel*.

Sabendo o ponto superior esquerdo de cada *pixel*, é possível obter um ponto no plano do *pixel* e a partir dele calcular a direção do raio. Se o mesmo ponto sempre for escolhido, a imagem gerada após a execução do *kernel* irá ficar com um aspecto serrilhado nas bordas dos objetos que compõem a cena. Para evitar isso, o ponto do plano é sempre escolhido de forma aleatória, sendo definido da forma:

$$P_r = P_{xy} + \vec{P_w} \cdot r_0 + \vec{P_h} \cdot r_1,$$

onde P_r é um ponto no plano do *pixel*, r_0 e r_1 são números reais no intervalo $[0,\ 1)$, que devem ser determinados de forma aleatória, utilizando um método de amostragem uniforme. Por fim, a direção do raio é calculada através da fórmula:

$$\vec{D} = \frac{P_r - O}{\|P_r - O\|},$$

onde $\vec{D}$ é um vetor unitário que representa a direção do raio, e O é a posição da câmera no espaço, que também é a origem do raio.

4.5 Percorrimento da Cena

O objetivo do algoritmo de percorrimento da cena é determinar qual é o triângulo mais próximo da origem de um determinado raio (na direção do mesmo). Na prática, isto significa percorrer a hierarquia da BVH, realizando testes de intersecção entre o raio e a AABB de cada nó. Se um raio atravessa a AABB de um nó, este raio também é testado com as AABBs de seus nós filhos. Este processo continua até que a AABB de um nó folha seja atingida. Quando a AABB de um nó folha é atingida, testes de interseção são realizados entre o raio e os triângulos que a AABB delimita, com o objetivo de determinar qual deles é o mais próximo, e também a distância t entre a origem do raio e o ponto de intersecção. A figura 25 mostra o exemplo de um raio percorrendo a cena (lado esquerdo) e a hierarquia da BVH (lado direito).

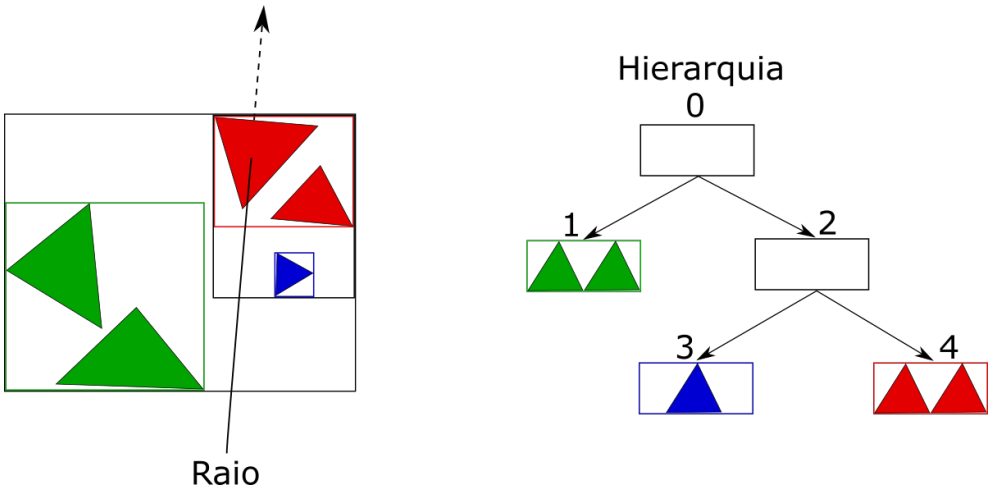


Figura 25: Exemplo de raio percorrendo a cena.

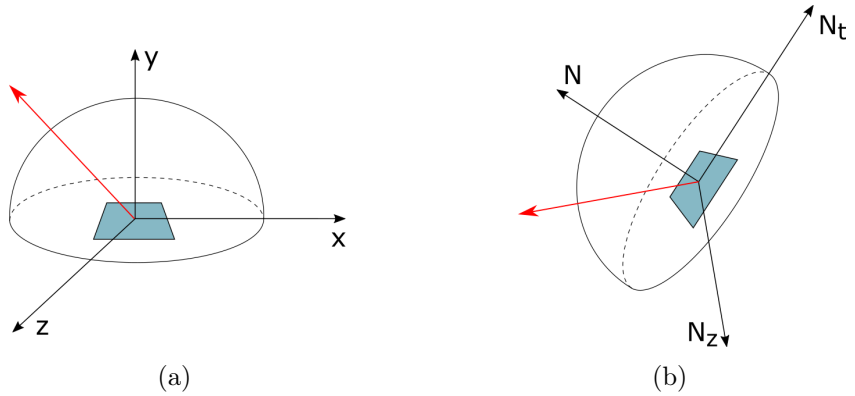
É possível notar pelo exemplo da figura 25, que o raio realiza testes de intersecção com as AABBs dos nós 0, 2 e 4, até finalmente realizar testes de intersecção com os dois triângulos vermelhos do nó 4 (nó folha). Como os testes de intersecção com os nós 1 e 3 falham, não é necessário realizar testes de intersecção com os triângulos contidos neles.

O algoritmo de intersecção raio-AABB foi implementado com base no algoritmo proposto por Aila, Laine e Karras (2012). Este algoritmo simplesmente determina se um raio atravessa ou não uma determinada AABB. Já o algoritmo de intersecção raio-triângulo foi implementado com base no algoritmo proposto por Möller e Trumbore (1997). Este algoritmo é capaz de determinar se o raio atravessa ou não um triângulo, e também determinar a distância t entre a origem do raio e o ponto de intersecção. Além disso, o algoritmo também determina as coordenadas baricêntricas do ponto de interseção em relação ao vértices dos triângulo. Estas coordenadas são utilizadas para calcular dados do ponto de intersecção, como o vetor normal e as coordenadas de textura.

4.6 Construção dos Raios Secundários

Quando um raio atinge alguma superfície da cena, um raio secundário é criado. Determinar a origem do raio secundário é simples, pois consiste exatamente no ponto de interseção do raio anterior com a superfície. Em geral, para determinar a direção do raio secundário, é preciso levar em consideração tanto o ângulo de incidência do raio anterior, quanto o vetor normal e o material da mesma. Como neste trabalho são utilizadas apenas cenas com superfícies difusas, a direção de saída dos raios é escolhida de maneira uniforme no hemisfério do mesmo sentido do vetor normal da superfície (figura 26b).

Para determinar a direção do raio secundário, é preciso seguir três passos: primeiro, determinar com uma probabilidade uniforme, uma direção aleatória no hemisfério para um vetor normal sobre o eixo y do sistema de coordenadas cartesianas (figura 26a). Em seguida, construir um novo sistema de coordenadas utilizando o vetor normal da superfície desejada. Por fim, converter as componentes do vetor obtido no primeiro passo para o novo sistema de coordenadas (figura 26b). Estas componentes formam exatamente a direção do raio secundário no sistema de coordenadas cartesianas. A figura 26 mostra a direção do raio no sistema de coordenadas cartesianas e a direção do mesmo no novo sistema de coordenadas construído utilizando o vetor normal da superfície.



(a) (b)

Figura 26: Sistemas de coordenadas: (a) Cartesianas, (b) Novo sistema.

4.7 Algoritmo de Path Tracing

A implementação do algoritmo de *path tracing* é realizada utilizado como base a equação de *rendering* formulada no espaço dos caminhos. Esta equação é mais conveniente, pois ela possui somatórios e produtórios que podem ser facilmente implementados como *loops* iterativos.

A equação de *rendering* formulada no espaço dos caminhos e estimada com o método de integração de Monte Carlo, já foi apresentada na seção 2. Para facilitar a leitura do leitor, ela será mostrada novamente nesta seção:

$$L_p = \frac{1}{N} \sum_{i=1}^{N} L_{pi}, \tag{26}$$

onde L_p é a radiância estimada para um *pixel* tomando uma média de N amostras, e L_{pi} é a radiância estimada de uma única amostra, escrita da forma:

$$L_{pi} = \sum_{s=0}^{N_s} C(s), \tag{27}$$

onde N_S é o tamanho máximo definido para o caminho, s é o tamanho do caminho a cada iteração e $C(s)$ é a contribuição de um único caminho de tamanho s, escrita como:

$$\begin{aligned} C(s) = \frac{L_e(y_0, \overrightarrow{y_0 x_s})}{p(y_0)} \; f_s(x_s, \overrightarrow{x_s y_0}, \overrightarrow{x_s x_{s-1}}) \; G(y_0 \leftrightarrow x_s) \\ \left(\prod_{i=1}^{s-1} \frac{f_s(x_i, \overrightarrow{x_i x_{i+1}}, \overrightarrow{x_i x_{i-1}}) \; |N_{x_i} \cdot \overrightarrow{x_i x_{i+1}}|}{p(\overrightarrow{x_i x_{i+1}})} \right). \end{aligned} \tag{28}$$

Como o algoritmo calcula apenas uma amostra de radiância por *pixel* a cada execução, o que é de fato implementado são as equações 27 e 28. A equação 27 é implementada simplesmente como um *loop* iterativo que é executado no intervalo de $s = 0$ até $s = N_s$, onde a cada iteração, o valor da contribuição $C(s)$ é acumulado em uma variável auxiliar. Já a equação 28 pode ser entendida da seguinte forma:

- A parte superior da equação, diz respeito a contribuição por amostragem direta da luz, ou seja, a radiância que sai de um ponto y_0 (da superfície da luz) e incide sobre um ponto x_s de alguma superfície;
- A parte inferior da equação (o produtório), diz respeito a radiância que é refletida por x_s e interage com outras superfícies da cena até atravessar a câmera no ponto x_0, através de um caminho arbritrário:

$$\overline{p}_s = x_s, x_{s-1}, x_{s-2}, ..., x_2, x_1, x_0.$$

A figura 27 mostra o exemplo de um caminho de tamanho máximo igual a 3. As setas contínuas representam o caminho formado, enquanto as setas tracejadas representam as contribuições por amostragem direta da luz. Os pontos x_0, x_1, x_2 e x_3 são os pontos que o algoritmo deve determinar para formar o caminho, onde para este caso $x_s = x_3$. O ponto y_0 é o ponto da luz em que cada ponto do caminho irá verificar se é iluminado diretamente por ele.

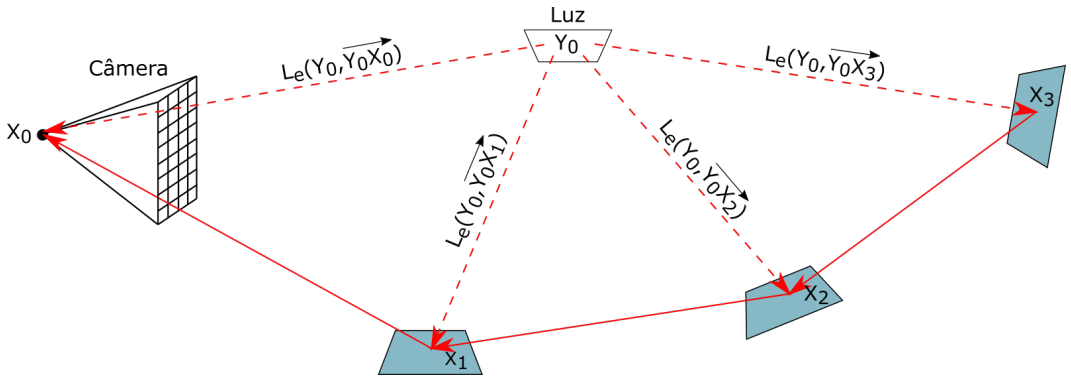


Figura 27: Caminho de tamanho máximo igual 3.

Na implementação da equação 28, inicialmente é realizada uma amostragem da superfície da luz para determinar o ponto y_0. A escolha deste ponto é realizada através do método proposto por Shirley, Wang e Zimmerman (1996), que é capaz de escolher um ponto aleatório na superfície de um triângulo de área A, com uma probabilidade:

$$p(y_0) = \frac{1}{A}.$$

Após a determinar o ponto y_0, são determinados os pontos que formam o caminho de tamanho s. Este caminho é construído utilizando o método *backward tracing*, ou seja, com as setas no sentido contrário as da figura 27. O ponto x_0 possui sempre o mesmo valor da posição da câmera. Para determinar o ponto x_1, primeiro constrói-se o raio primário para determinar os valores de O e $\vec{D}$. Em seguida, é realizado o percorrimento da cena para determinar a distância t entre a posição da câmera (x_0) e a superfície mais próxima. Uma vez determinados os valores de O, $\vec{D}$ e t, o ponto x_1 é obtido da forma:

$$x_1 = O + t \cdot \vec{D}.$$

Para determinar o ponto x_2, um raio secundário é construído com a origem igual ao ponto x_1 e com a direção $\vec{D_2}$ determinada da maneira como foi mostrada na subseção 4.6. Após construir este raio, a cena é percorrida para determinar o valor de t_2 (distância entre o ponto x_1 e a superfície mais próxima). Finalmente, o ponto x_2 é obtido da forma:

$$x_2 = x_1 + t_2 \cdot \vec{D_2}.$$

O mesmo processo utilizado para determinar x_2 se repete até que seja determinado o ponto x_s. Durante todo o processo de determinação dos pontos que formam o caminho, os valores que são utilizados na equação 28, como ângulos, vetores, BSDFs e PDFs, são armazenados em um vetor de variáveis auxiliares. Após obter todos esses valores, o algoritmo calcula o produtório através de um *loop* iterativo que vai de $i = 1$ até $i = s - 1$.

Uma vez determinado o caminho da luz e calculado o produtório, é preciso verificar se os pontos x_s e y_0 são mutuamente visíveis, se isto ocorrer, significa que o ponto x_s é iluminado diretamente por y_0. A função responsável por determinar isto é a função $V(y_0 \leftrightarrow x_s)$, que faz parte da função $G(y_0 \leftrightarrow x_s)$. A função $V(y_0 \leftrightarrow x_s)$ é implementada construindo um raio com origem $O = x_s$ e direção:

$$\vec{D} = \frac{y_0 - x_s}{\|y_0 - x_s\|}.$$

Em seguida, este raio percorre a cena para determinar o triângulo mais próximo. Os pontos y_0 e x_s serão mutuamente visíveis se o triângulo mais próximo for exatamente o mesmo triângulo da luz que possui o ponto y_0.

4.8 Algoritmo de Path Tracing Bidirecional

A implementação do algoritmo de *path tracing* bidirecional é muito semelhante à implementação do algoritmo mostrado na subseção 4.7. Enquanto o algoritmo mostrado na subseção 4.7 determina um caminho e verifica se cada ponto é mutuamente visível com único ponto da luz, o algoritmo bidirecional determina dois caminhos e verifica se os pontos deles são mutuamente visíveis entre si. Se dois pontos de caminhos distintos forem mutuamente visíveis, um terceiro caminho é formado a partir da conexão entre eles. Nesta seção esses caminhos serão chamados de sub-caminhos, que são apresentados a seguir:

- **O sub-caminho da câmera:** Este sub-caminho é construído utilizando o método *backward tracing*, exatamente da mesma forma como foi mostrada na subseção 4.7, ou seja, partindo da posição da câmera.
- **O sub-caminho da luz:** Este sub-caminho é semelhante ao sub-caminho da câmera, porém é construído utilizando o método *forward tracing*, ou seja, partindo de um ponto aleatório da superfície da luz.

A figura 28 mostra um exemplo dos sub-caminhos da câmera e da luz. As setas vermelhas representam o sub-caminho da câmera, enquanto as setas azuis representam o sub-caminho da luz. As linhas tracejadas representam as possíveis conexões entre os vértices dos sub-caminhos, formando novos caminhos.

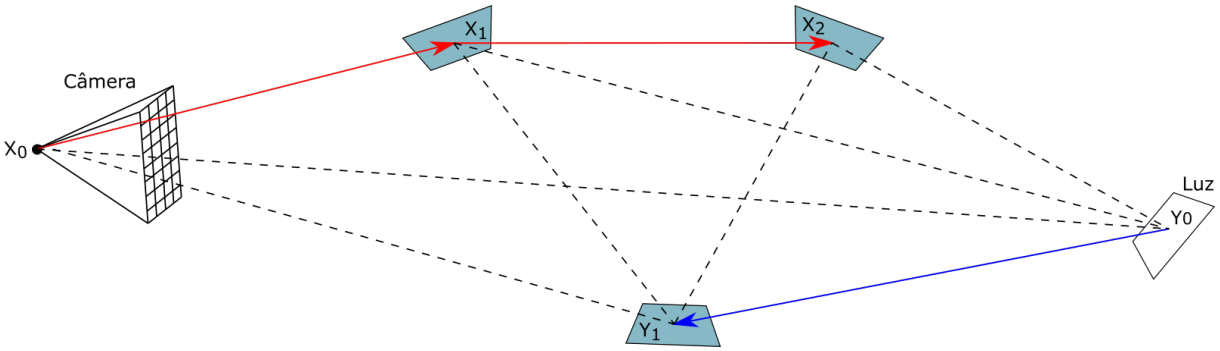


Figura 28: Sub-caminhos da câmera (vermelho) e da luz (azul).

Além das diferenças entre os pontos iniciais dos sub-caminhos, também existem diferenças em relação ao primeiro raio construído para cada um deles. Enquanto para o sub-caminho da câmera o primeiro raio utilizado é sempre o raio primário, no sub-caminho da luz, a direção do primeiro raio é definida de forma aleatória, semelhante à construção de raios secundários mostrada na subseção 4.6.

Como já era de se esperar, a implementação do algoritmo bidirecional também usa como base a equação de *rendering* no espaço dos caminhos, pelos mesmos motivos apresentados na subseção anterior. A equação de *rendering* formulada para o método bidirecional e estimada com o método de Monte Carlo é mostrada novamente a seguir:

$$L_p = \frac{1}{N} \sum_{i=0}^{N} L_{pi}, \tag{29}$$

onde L_p é a radiância estimada para um *pixel* tomando uma média de N amostras, e L_{pi} é a radiância estimada de uma única amostra, escrita da forma:

$$L_{pi} = \sum_{s=0}^{N_S} \sum_{t=0}^{N_T} W(s,t)\, C(s,t), \tag{30}$$

onde N_S e N_T são respectivamente os tamanhos máximos definidos para os sub-caminhos da câmera e da luz. $C(s,t)$ é a função que mede a contribuição do novo caminho formado pela conexão dos vértices x_s e y_t. A função $W(s,t)$ é uma função que mede o peso de cada contribuição $C(s,t)$. A função $C(s,t)$ para o caso mais geral é mostrada a seguir:

$$\begin{aligned} C(s,t) = \frac{L_e(y_0, \overrightarrow{y_0 y_1})}{p(y_0, \overrightarrow{y_0 y_1})} \left(\prod_{j=1}^{t-1} \frac{f_s(y_j, \overrightarrow{y_j y_{j-1}}, \overrightarrow{y_j y_{j+1}})\ \left| N_{y_j} \cdot \overrightarrow{y_j y_{j+1}} \right|}{p(\overrightarrow{y_j y_{j+1}})} \right) \\ f_s(y_t, \overrightarrow{y_t y_{t-1}}, \overrightarrow{y_t x_s})\ G(y_t \leftrightarrow x_s)\ f_s(x_s, \overrightarrow{x_s y_t}, \overrightarrow{x_s x_{s-1}}) \\ \left(\prod_{i=1}^{s-1} \frac{f_s(x_i, \overrightarrow{x_i x_{i+1}}, \overrightarrow{x_i x_{i-1}})\ \left| N_{x_i} \cdot \overrightarrow{x_i x_{i+1}} \right|}{p(\overrightarrow{x_i x_{i+1}})} \right). \end{aligned} \tag{31}$$

Os dois somatórios da equação 30 são implementados através de dois *loops* iterativos que variam de $s = 0$ até $s = N_L$ e de $t = 0$ até $t = N_T$. A cada iteração, o resultado do produto entre $W(s,t)$ e $C(s,t)$ é acumulado em uma variável auxiliar.

A implementação da função $C(s,t)$ é feita de forma semelhante à da função $C(s)$ apresentada na subseção 4.7. Na equação 31, o produtório da parte superior diz respeito ao sub-caminho da luz, enquanto o produtório da parte inferior diz respeito ao sub-caminho da câmera. Durante a construção dos sub-caminhos, os dados referentes à ângulos, vetores, BSDFs e PDFs, são armazenados em um vetor de variáveis auxiliares para serem utilizados no cálculo dos produtórios e dos demais elementos da equação.

O termo $p(y_0, \overrightarrow{y_0 y_1})$ da função $C(s,t)$, representa a probabilidade de escolher y_1 como o próximo vértice do caminho da luz. O caminho inicia em um vértice y_0 pertencente à um triângulo de luz de área A e vai na direção de y_1 com a probabilidade:

$$p(y_0, \overrightarrow{y_0 y_1}) = \frac{1}{2\pi A}.$$

A regra geral da função $W(s,t)$, da equação 30, diz que a soma dos pesos dos caminhos com mesmo comprimento deve ser igual a 1. Analisando todos os possíveis caminhos formados através das conexões da figura 28, obtém-se a tabela 2, que mostra as informações referentes aos novos caminhos formados.

Tabela 2: Caminhos formados após as conexões dos vértices da figura 28.

Conexão	Caminho Formado	Comprimento
x_0 *e* y_0	x_0, y_0	1
x_0 *e* Y_1	x_0, y_1, y_0	2
x_1 *e* y_0	x_0, x_1, y_0	2
x_1 *e* y_1	x_0, x_1, y_1, y_0	3
x_2 *e* y_0	x_0, x_1, x_2, y_0	3
x_2 *e* y_1	x_0, x_1, x_2, y_1, y_0	4

É possível notar pela tabela 2, que existe apenas um caminho com comprimento igual a 1, formado pela conexão dos vértices x_0 e y_0, e apenas um caminho com comprimento igual a 4, formados pela conexão dos vértices x_2 e y_1. Os caminhos que se repetem são os que possuem comprimento 2 e 3, onde ocorrem duas vezes cada. Se adicionarmos mais vértices nos caminhos da câmera ou da luz e realizarmos novas conexões, seria possível perceber que os comprimentos de caminho mais longo e mais curto nunca se repetem, enquanto os demais comprimentos sempre ocorrerem duas vezes, de modo que a função $W(s,t)$ pode ser definida como:

$$W(s,t) = \begin{cases} 1, & \text{se } (s,t) = (0,0) \text{ ou } (s,t) = (N_S, N_T); \\ 0.5, & \text{caso contrário.} \end{cases}$$

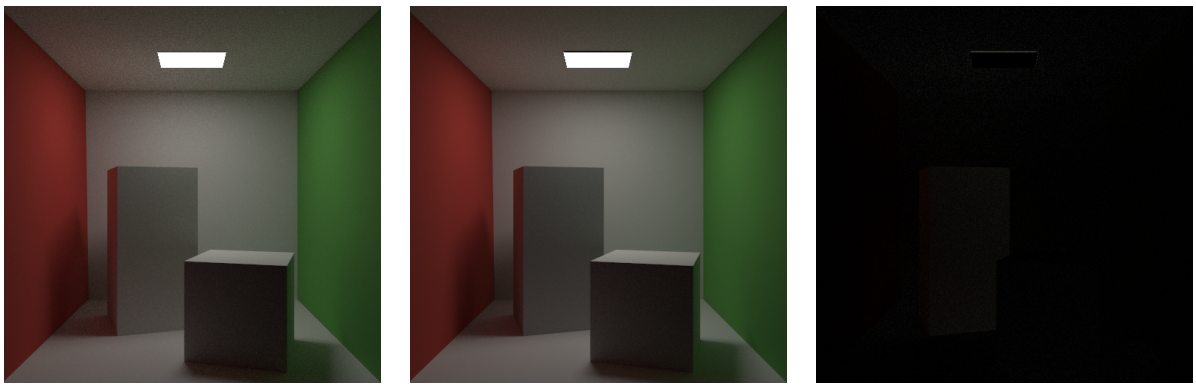
5 APRESENTAÇÃO E ANÁLISE DOS RESULTADOS

Esta seção apresenta os resultados obtidos após o desenvolvimento da aplicação. Os testes serão realizados da forma como foram propostos na seção 3. Primeiramente, serão comparados os resultados obtidos neste trabalho com os do *software Mitsuba Renderer* (JAKOB, 2010). Por fim, serão apresentados os resultados referentes a eficiência dos algoritmos implementados.

5.1 Testes de Qualidade dos Algoritmos

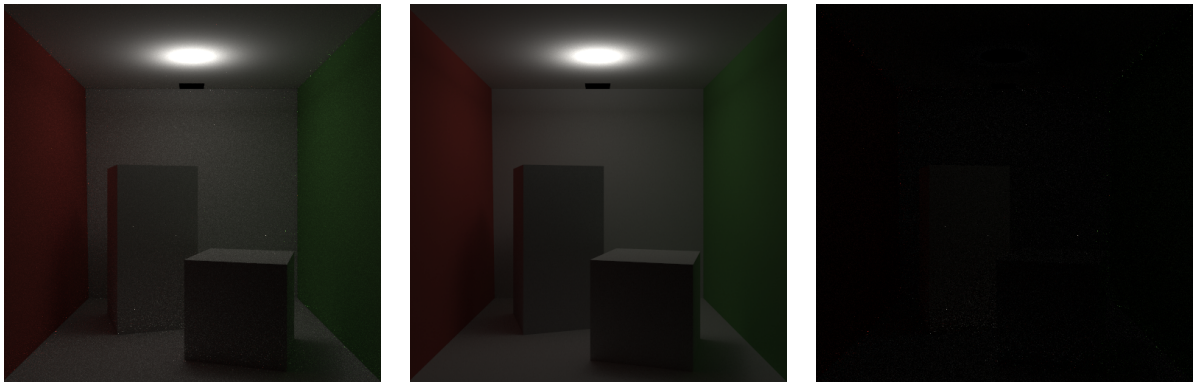
Para verificar a qualidade dos algoritmos, foram escolhidas duas cenas, onde para cada uma delas serão renderizadas duas imagens: uma utilizando o *software* desenvolvido neste trabalho e outra utilizando o *Mitsuba Renderer*. As imagens possuem a mesma resolução (512x512), renderizam a mesma região das cenas e possuem o mesmo número de amostras por *pixel* (256).

A cenas escolhidas para realizar as comparações foram a cena 1 (*Cornell Box 1*) e a cena 2 (*Cornell Box 2*). Estas cenas foram escolhidas por serem cenas clássicas para testes e por serem disponibilizas pelo próprio *Mitsuba Renderer*. A a cena 1 possui a maior parte da sua geometria iluminada diretamente e será utilizada para comparar os algoritmos de *path tracing* com amostragem direta da luz. A cena 2 por sua vez, possui a maior parte de geometria não iluminada diretamente e será utilizada para comparar os algoritmos que utilizam o método bidirecional. A figura 29 mostra a comparação entre os resultados obtidos para o algoritmo de *path tracing* com amostragem direta da luz e a figura 30 mostra a comparação dos resultados obtidos para o método bidirecional.



(a) (b) (c)

Figura 29: Comparação dos algoritmos de *path tracing* com amostragem direta da luz. (a) Este Trabalho, (b) *Mistuba Renderer*, (c) Diferença.



(a) (b) (c)

Figura 30: Comparação dos algoritmos de *path tracing* bidirecional. (a) Este Trabalho, (b) *Mistuba Renderer*, (c) Diferença.

É possível notar pelas figuras 29 e 30, que os resultados obtidos neste trabalho são bastante parecidos com os obtidos pelo *Mitsuba Renderer*. Características como as sombras do objetos e as regiões que recebem mais luz, são praticamente idênticas aos das imagens de referência em ambos os algoritmos. Os resultados divergem um pouco quando se considera o brilho das imagens, pois tanto para a cena 1, quanto para a cena 2, os resultados deste trabalho apresentam uma imagem com um pouco mais de brilho, principalmente na caixa maior que fica na parte de trás da *Cornell Box* (figura 29c). Além disso, a imagem gerada por este trabalho para cena 2 (figura 30a) apresenta um pouco mais de ruído do que a imagem de referência (figura 30b).

A partir destes testes é possível concluir que o resultados deste trabalho estão muito próximos do que seria o correto. Acredita-se que a diferenças de brilho e de ruído entre as imagens, ocorram devido a alguma etapa de pós-processamento realizado pelo *Mitsuba Renderer*. Infelizmente, as configurações que este *software* permite realizar, referentes ao pós-processamento, são bastante limitadas.

5.2 Resultados Estatísticos

Foram realizados testes para verificar a eficiência dos algoritmos implementados. As tabelas 3, 4, 5, 6, 7, 8 mostram os resultados para todas as 6 cenas que foram apresentadas na subseção 3.5. A 1ª coluna de cada tabela, indica o método utilizado pelo algoritmo de *path tracing*, que são três: o original (sem amostragem), o método com amostragem direta da luz e o método bidirecional. A 2ª coluna indica o número de amostras por *pixel* utilizadas. A 3ª Coluna indica o tempo médio gasto para calcular uma única amostra para cada um dos *pixels* da imagem. A 4ª e última coluna indica a eficiência do método utilizado, onde o seu valor indica o percentual de amostras que conseguiram obter êxito ao encontrar um caminho entre a câmera e a luz.

Tabela 3: Resultados estatísticos da cena 1.

Método	Nº de Amostras	Tempo Médio	Eficiência
Original	200	61,0 ms	2,22%
Amostragem Direta da Luz	200	90,8 ms	84,6%
Bidirecional	200	152,5 ms	91,1%

Tabela 4: Resultados estatísticos da cena 2.

Método	Nº de Amostras	Tempo Médio	Eficiência
Original	200	61,4 ms	0,34%
Amostragem Direta da Luz	200	67,9 ms	52,6%
Bidirecional	200	143,3 ms	88,2%

Tabela 5: Resultados estatísticos da cena 3.

Método	Nº de Amostras	Tempo Médio	Eficiência
Original	200	57,7 ms	0,06%
Amostragem Direta da Luz	200	84,9 ms	6,36%
Bidirecional	200	154,8 ms	87,6%

Tabela 6: Resultados estatísticos da cena 4.

Método	Nº de Amostras	Tempo Médio	Eficiência
Original	200	470,1 ms	12,2%
Amostragem Direta da Luz	200	701,2 ms	81,1%
Bidirecional	200	1009,2 ms	82,1%

Tabela 7: Resultados estatísticos da cena 5.

Método	Nº de Amostras	Tempo Médio	Eficiência
Original	200	489,1 ms	0,32%
Amostragem Direta da Luz	200	582,0 ms	57,6%
Bidirecional	200	897,5 ms	68,1%

Tabela 8: Resultados estatísticos da cena 6.

Método	Nº de Amostras	Tempo Médio	Eficiência
Original	200	391,2 ms	0,6%
Amostragem Direta da Luz	200	549,7 ms	48,6%
Bidirecional	200	862,9 ms	58,3%

5.3 Renderizações e Mapas de Calor

As figuras 31, 32, 33, 34, 35 e 36 mostram as renderizações e seus mapas de calor. As figuras estão organizadas da seguinte forma: A 1ª coluna de cada figura mostra o resultado do método original, a 2ª coluna mostra o resultado do método de amostragem direta da luz e a 3ª coluna mostra o resultado do bidirecional. A 1ª linha de cada imagem mostra as renderizações, enquanto a 2ªlinha mostra os respectivos mapas de calor.

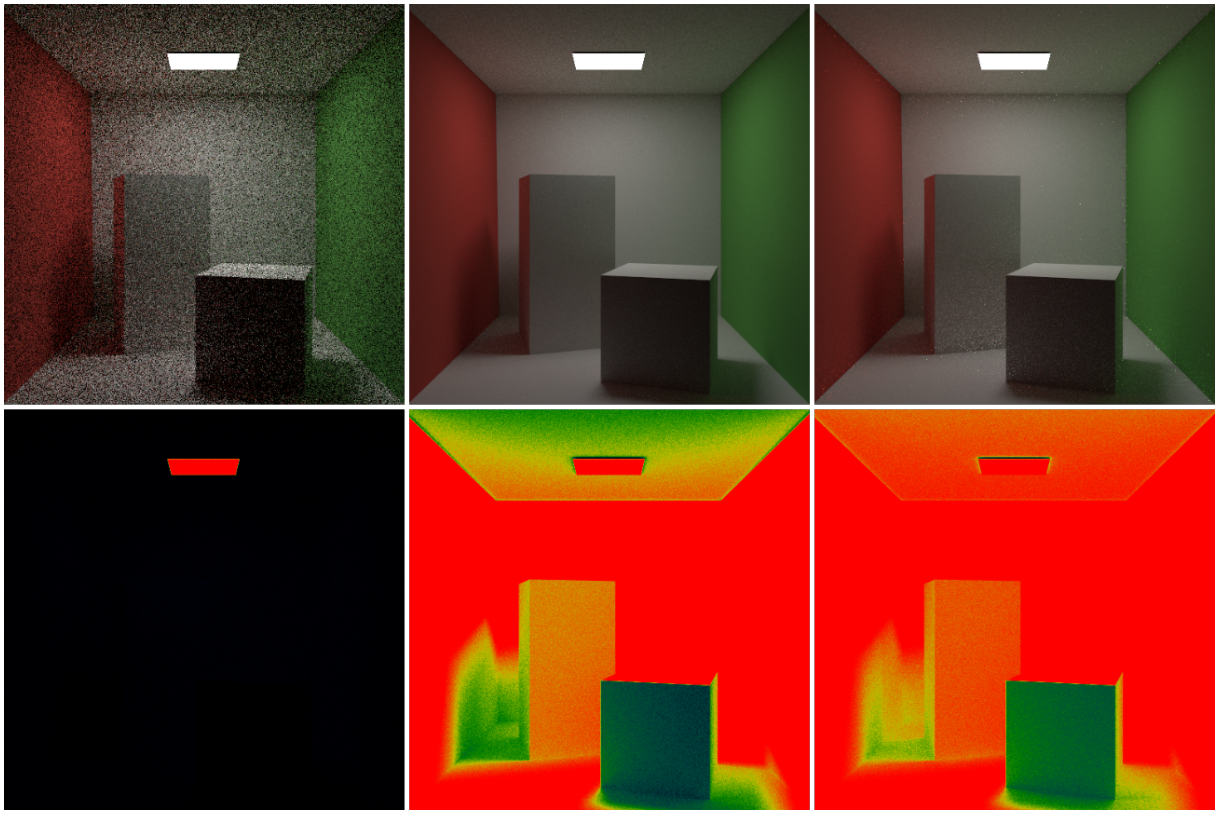


Figura 31: Renderizações e mapas de calor da cena 1.

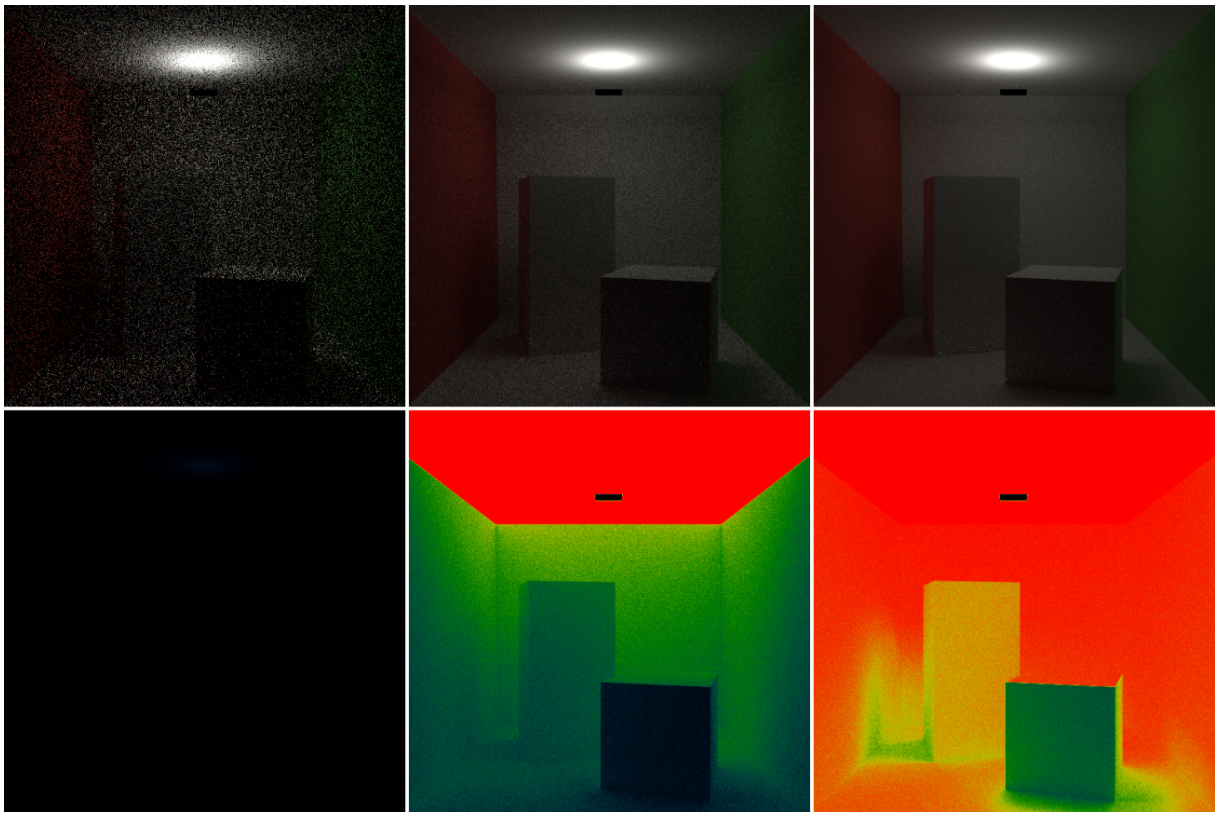


Figura 32: Renderizações e mapas de calor da cena 2.

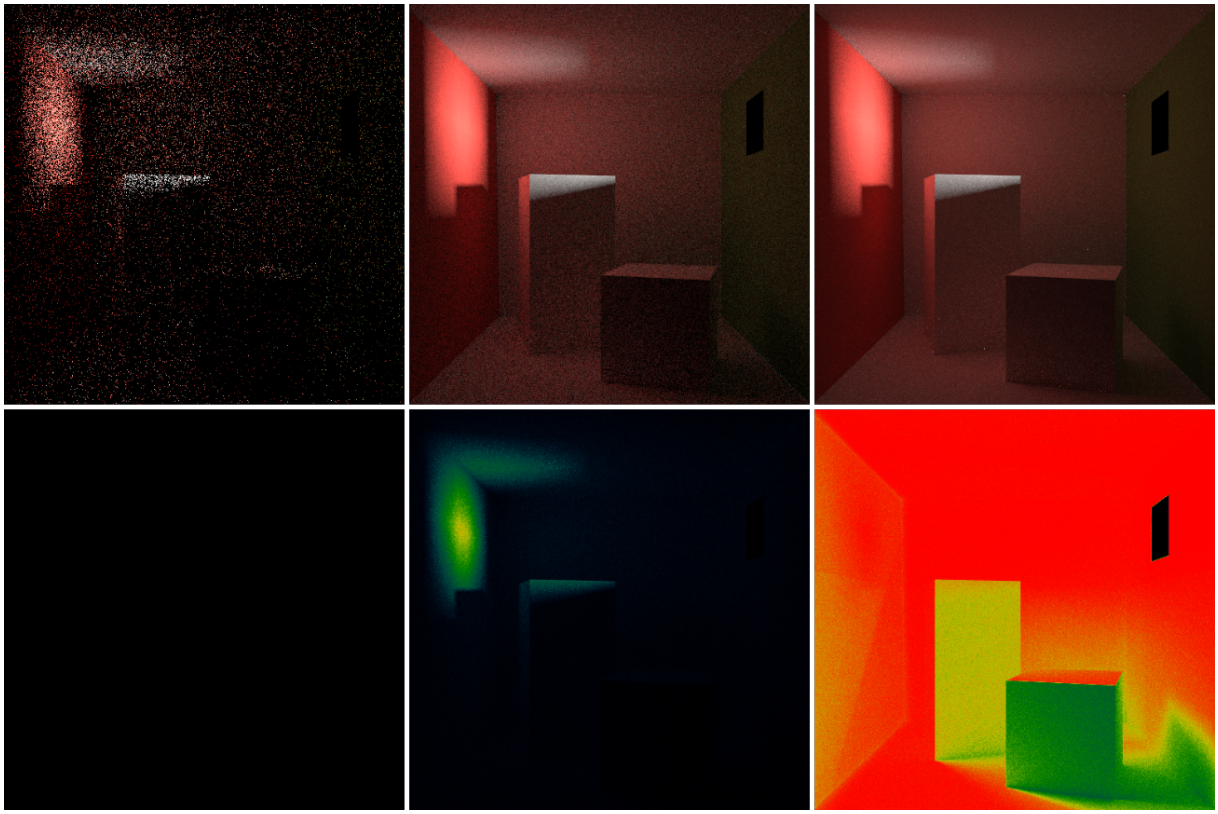


Figura 33: Renderizações e mapas de calor da cena 3.

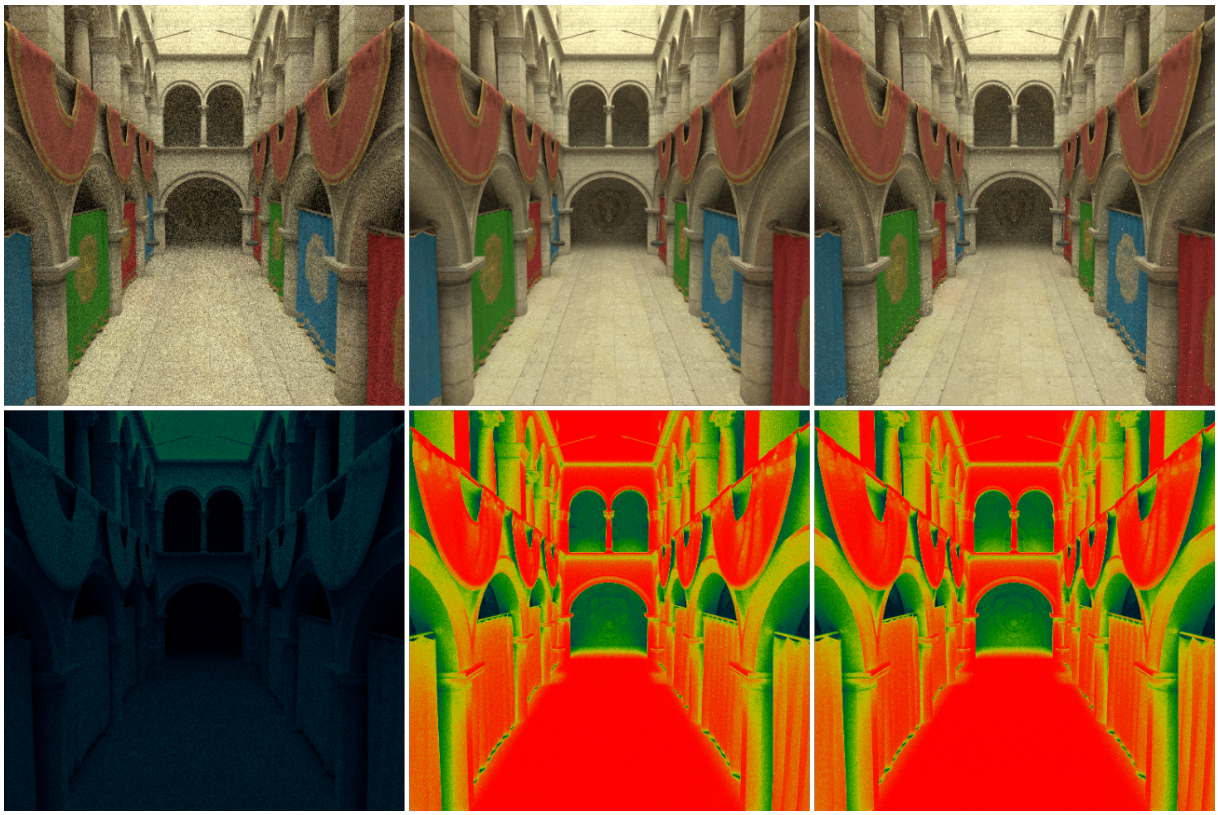


Figura 34: Renderizações e mapas de calor da cena 4.

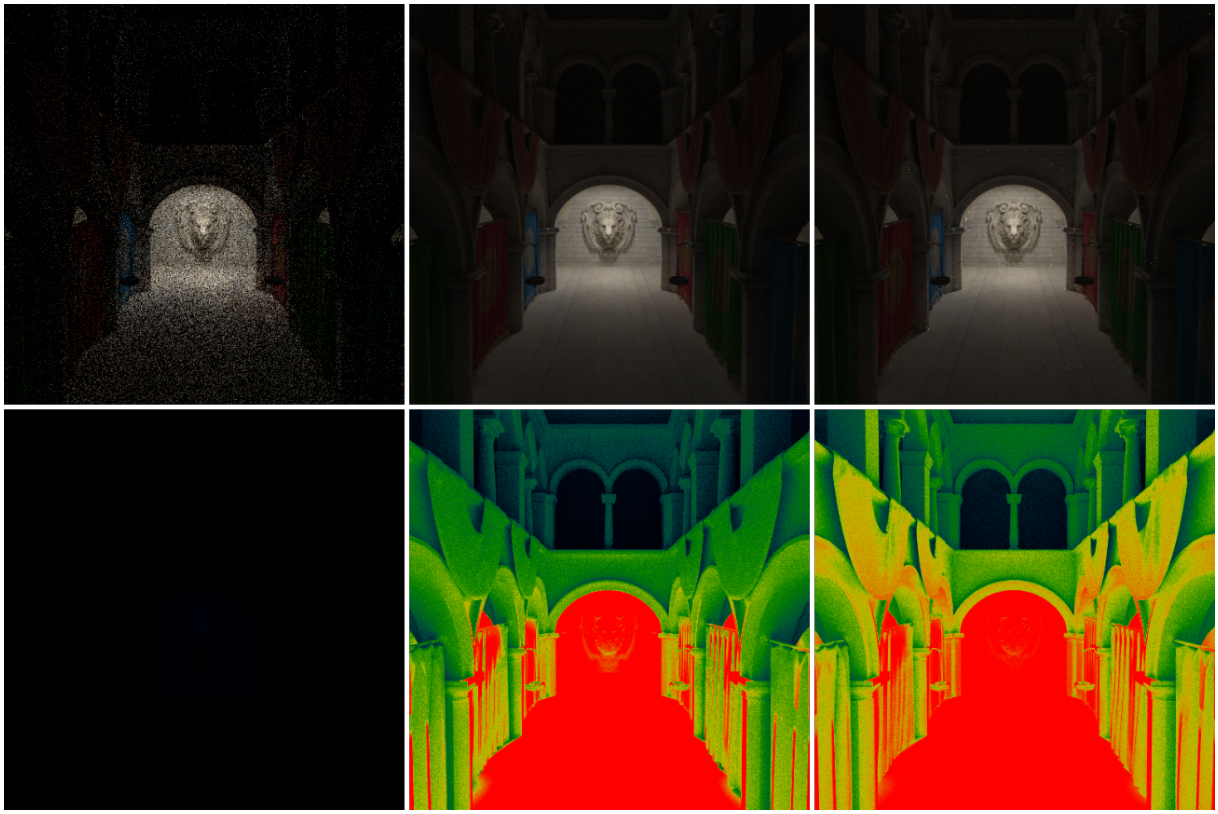


Figura 35: Renderizações e mapas de calor da cena 5

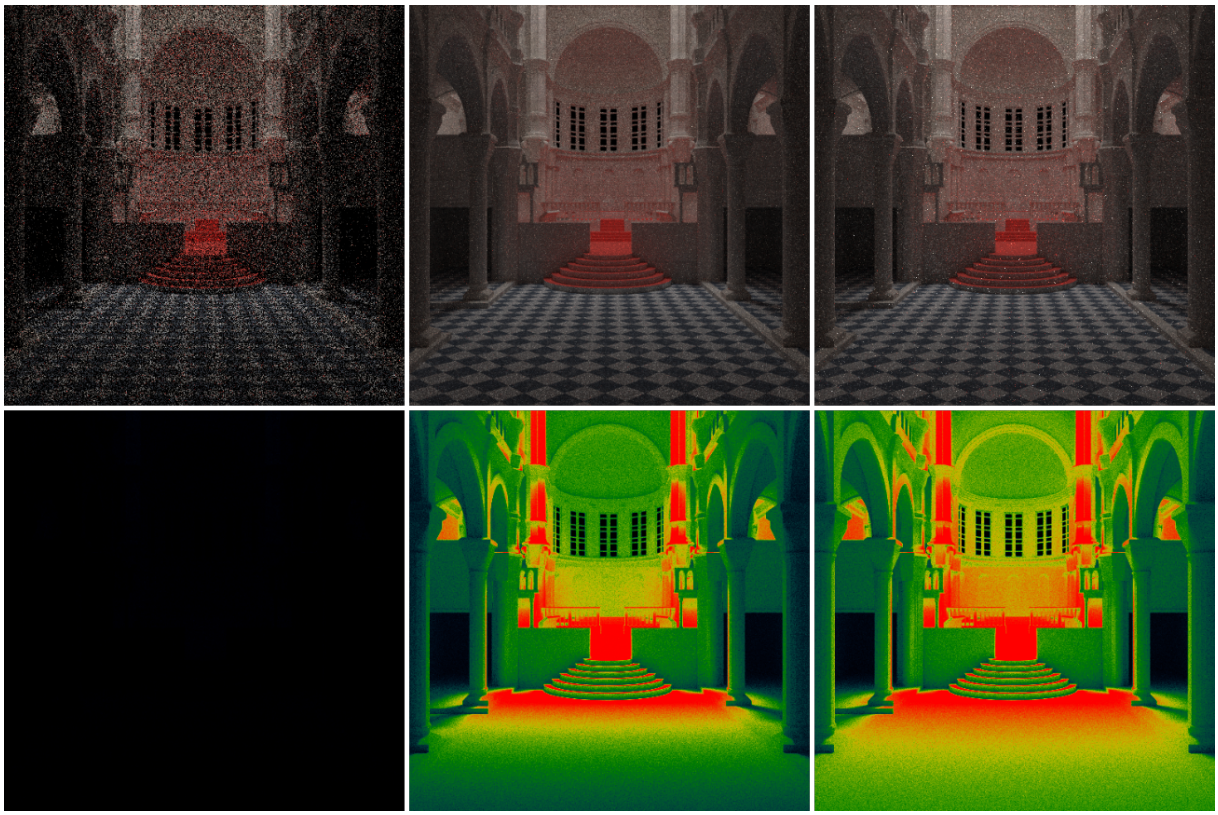


Figura 36: Renderizações e mapas de calor da cena 6

5.4 Análise de Resultados

Os resultados mostram que algoritmo bidirecional obteve melhores resultados que os outros dois em todas as cenas, quando se considera a eficiência ao encontrar caminhos entre a câmera e a fonte de luz. O melhor resultado ocorreu na cena 3, onde o algoritmo teve uma eficiência de $87,6\%$ contra apenas $6,36\%$ do método de amostragem direta da luz (tabela 5). Pode-se notar que os resultados obtidos para as cenas 2 e 3 (do método bidirecional), mostram imagens com menos ruído do que as geradas pelos outros métodos (figuras 32 e 33). Estas cenas são exatamente as que possuem a maior parte da geometria não iluminada diretamente pela fonte de luz. Apesar dos bons resultados, o algoritmo bidirecional apresenta tempos execução bastante elevados quando comparados aos outros dois algoritmos. Um dos piores resultados é o da cena 2, onde ele custou um pouco mais do dobro do tempo para executar, quando comparado aos demais algoritmos (tabela 4).

Por sua vez, o algoritmo que utiliza o método de amostragem direta da luz obteve resultados melhores que os do algoritmo original em todas as cenas, com diferenças de consideravelmente grandes, quando se considera a eficiência ao encontrar a luz. Para as cenas que possuem a maior parte da geometria iluminada diretamente (cenas 1 e 4), o algoritmo que utiliza o método de amostragem direta obteve resultados muito próximos aos do bidirecional (tabelas 3 e 6). Porém, para estes casos ele obteve os resultados com aproximadamente 40% menos tempo do que o algoritmo bidirecional, demostrando que ele é mais viável quando utilizado em cenas com estas características.

O algoritmo original apresenta resultados com imagens muito ruidosas, devido ao fato de ter mais dificuldade em conseguir encontrar caminhos entre a câmera e a luz, fazendo com que a maioria dos *pixels* possuam uma cor preta. Mesmo com a expressiva ineficiência que este algoritmo apresenta, ainda é possível em alguns casos, como o da cena 4 (figura 34), obter imagens ainda comparáveis com as das outras técnicas. Isto só é possível quando a fonte de luz é relativamente grande em relação a cena, fazendo com que seja mais fácil para este algoritmo encontrar a luz.

É possível notar pelos resultados das imagens de calor, que as regiões de sombras são as que possuem mais dificuldades para encontrar caminhos até a luz. Esta situação fica mais evidente quando observado os mapas de calor gerados pelo método de amostragem direta para as cenas 2 e 3 (figuras 32 e 33 respectivamente), onde a maior parte destas cenas consistem em regiões de sombra. Analisando os resultados do algoritmo bidirecional para estas cenas, é possível notar uma melhora de eficiência significativa, no entanto, as regiões de sombra ainda são as que apresentam os piores resultados.

6 CONCLUSÕES E TRABALHOS FUTUROS

Os resultados mostraram que o algoritmo de *path tracing* bidirecional consegue encontrar caminhos entre a câmera e a luz com mais eficiência que os demais algoritmos, proporcionando resultados melhores quando utilizado o mesmo número de amostras por *pixel*. Em situações específicas, onde a luz é de difícil acesso para a maior parte da geometria da cena, o algoritmo demonstrou ser até 13x mais eficiente que o algoritmo que utiliza amostragem direta da luz, com um custo de tempo aproximadamente 2x maior. Com isso, o algoritmo demostra ser extremamente viável para estas situações. Porém, quando estas situações não ocorrem, o algoritmo não é tão eficiente, possuindo resultados muito próximos dos demais métodos, o que torna a sua utilização inviável devido ao grande custo de tempo que ele leva para executar.

Uma das maiores dificuldades encontradas durante a realização deste trabalho, foi sem dúvidas a compreensão dos conceitos radiométricos e da equação de *rendering*. Os resultados iniciais ficaram um pouco distantes daquilo que se era esperado devido a falta de compreensão inicial. O que fez com que fosse gasto um pouco mais de tempo até que fossem obtidos resultados considerados corretos. Após o término deste trabalho, posso afirmar que compreendo com mais detalhes os conceitos que envolvem a radiometria, assim como todos os elementos que constituem a equação de *rendering*, nas suas diversas formulações.

Uma outra grande dificuldade foi a implementação dos algoritmos para serem executados em GPU, pois trata-se de um *hardware* muito diferente e com várias limitações de programação (quando comparado a uma CPU). As APIs utilizadas para a programação de GPUs, fornecem mecanismos bastante limitados para a depuração de códigos fonte, fazendo com que a solução de erros e problemas ocorridos durante as implementações tivessem um custo maior de tempo. Estes problemas afetaram principalmente a etapa de serializarão da BVH.

Um das possíveis melhorias que poderiam ser realizas no futuro, é a implementação de superfícies com outros tipos de materiais, como vidro, metais, espelhos, entre outros, com o objetivo de aumentar o realismo das imagens geradas. Também poderiam ser implementadas técnicas de pós-processamento, de modo que pudessem ser geradas imagens com menos ruído, mesmo para os algoritmos que demostraram ser menos eficientes. Novas estruturas de aceleração também poderiam ser implementadas, de modo a permitir a verificação de quais delas são a mais eficientes para os diferentes tipos de cenas.

REFERÊNCIAS

ADAMSEN, M. *Bidirectional Path Tracing*. 2009. <http://www.maw.dk/?page_id=78>. Acessado em: 17-10-2018.

AILA, T.; LAINE, S.; KARRAS, T. Understanding the efficiency of ray traversal on gpus–kepler and fermi addendum. *NVIDIA Corporation, NVIDIA Technical Report NVR-2012-02*, 2012.

AZEVEDO, E.; CONCI, A. *Computação gráfica: teoria e prática*. [S.l.]: Elsevier, 2003.

COHEN, M. F.; GREENBERG, D. P. The hemi-cube: A radiosity solution for complex environments. In: ACM. *ACM SIGGRAPH Computer Graphics*. [S.l.], 1985. v. 19, n. 3, p. 31–40.

COOK, S. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN 9780124159334, 9780124159884.

DUTRE, P.; BEKAERT, P.; BALA, K. *Advanced global illumination*. [S.l.]: AK Peters/CRC Press, 2006.

FLEURY, A.; NAKANO, D.; CORDEIRO, J. Mapeamento da indústria brasileira e global de jogos digitais. *Retirado de https://goo. gl/kE7mA3*, 2014.

GORAL, C. M. et al. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph.*, ACM, New York, NY, USA, v. 18, n. 3, p. 213–222, jan. 1984. ISSN 0097-8930.

IZE, T.; WALD, I.; PARKER, S. G. Ray tracing with the bsp tree. In: *2008 IEEE Symposium on Interactive Ray Tracing*. [S.l.: s.n.], 2008. p. 159–166.

JAKOB, W. *Mitsuba renderer*. 2010. <https://www.mitsuba-renderer.org/>. Acessado em: 27-02-2019.

KAJIYA, J. T. The rendering equation. *SIGGRAPH Comput. Graph.*, ACM, New York, NY, USA, v. 20, n. 4, p. 143–150, ago. 1986. ISSN 0097-8930.

KELLER, A. et al. The path tracing revolution in the movie industry. In: ACM. *ACM SIGGRAPH 2015 Courses*. [S.l.], 2015. p. 24.

LAFORTUNE, E. P.; WILLEMS, Y. D. Bi-directional path tracing. In: . [S.l.: s.n.], 1993. p. 145–153.

LAINE, S.; KARRAS, T.; AILA, T. Megakernels considered harmful: wavefront path tracing on gpus. In: ACM. *Proceedings of the 5th High-Performance Graphics Conference*. [S.l.], 2013. p. 137–143.

MCGUIRE, M. *Computer Graphics Archive*. 2017. <https://casual-effects.com/data>. Acessado em: 27-02-2019.

MISIC, M.; DURDEVIC, D.; TOMASEVIC, M. Evolution and trends in gpu computing. In: . [S.l.: s.n.], 2012. p. 289–294. ISBN 978-1-4673-2577-6.

MöLLER, T.; TRUMBORE, B. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, A. K. Peters, Ltd., Natick, MA, USA, v. 2, n. 1, p. 21–28, out. 1997. ISSN 1086-7651.

MUNSHI, A. et al. *OpenCL Programming Guide*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2011. ISBN 0321749642, 9780321749642.

PADRÃO, B. N.; ALIPRANDI, D. C.; PORTO, J. F. O mercado de nicho da netflix e a exclusão no entretenimento digital. 2018.

PHARR, M.; JAKOB, W.; HUMPHREYS, G. *Physically based rendering: From theory to implementation*. [S.l.]: Morgan Kaufmann, 2016.

PURCELL, T. J. et al. Ray tracing on programmable graphics hardware. In: ACM. *ACM SIGGRAPH 2005 Courses*. [S.l.], 2005. p. 268.

RESHETOV, A.; SOUPIKOV, A.; HURLEY, J. Multi-level ray tracing algorithm. *ACM Trans. Graph.*, ACM, New York, NY, USA, v. 24, n. 3, p. 1176–1185, jul. 2005. ISSN 0730-0301.

SHIRLEY, P.; WANG, C. Direct lighting calculation by monte carlo integration. In: *Photorealistic Rendering in Computer Graphics*. [S.l.]: Springer, 1994. p. 52–59.

SHIRLEY, P.; WANG, C.; ZIMMERMAN, K. Monte carlo techniques for direct lighting calculations. *ACM Transactions on Graphics (TOG)*, ACM, v. 15, n. 1, p. 1–36, 1996.

VEACH, E. *Robust monte carlo methods for light transport simulation*. [S.l.]: Stanford University PhD thesis, 1997.

WALD, I.; BOULOS, S.; SHIRLEY, P. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, ACM, New York, NY, USA, v. 26, n. 1, jan. 2007. ISSN 0730-0301.

WALD, I. et al. Ray tracing animated scenes using coherent grid traversal. *ACM Trans. Graph.*, ACM, New York, NY, USA, v. 25, n. 3, p. 485–493, jul. 2006. ISSN 0730-0301.

WHITTED, T. An improved illumination model for shaded display. *Commun. ACM*, ACM, New York, NY, USA, v. 23, n. 6, p. 343–349, jun. 1980. ISSN 0001-0782.